TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

SYNCHRONIZACE DATABÁZE MOBILNÍHO ZAŘÍZENÍ OS ANDROID SE SERVEROVOU DATABÁZÍ

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Leoš Dostál**
Vedoucí práce: Ing. Přemysl Svoboda

Liberec 2014

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

SYNCHRONIZATION OF AN ANDROID OS MOBILE DEVICE DATABASE WITH A SERVER DATABASE

Diploma thesis

Study programme: N2612 – Electrical Engineering and Informatics
Study branch: 1802T007 – Information Technology

Author: **Bc. Leoš Dostál**
Supervisor: Ing. Přemysl Svoboda

Liberec 2014

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2013/2014**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Leoš Dostál**
Osobní číslo:	**M12000200**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Synchronizace databáze mobilního zařízení OS Android se serverovou databází**
Zadávající katedra:	**Ústav mechatroniky a technické informatiky**

Zásady pro vypracování:

1. Navrhněte způsob synchronizace databáze mobilního zařízení se serverovou databází.
2. Naprogramujte synchronizaci dle bodu 1) ve formě knihovny v jazyce Java pro OS Android.
3. Naprogramujte synchronizaci dle bodu 1) pro serverovou část na platformě .NET.
4. Proveďte test synchronizace pomocí ukázkové aplikace na OS Android a webové aplikace v ASP.NET.

Rozsah grafických prací: **dle potřeby dokumentace**

Rozsah pracovní zprávy: **40–50 stran**

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

[1] **Programování v jazyce Java, David Flanagan, Praha : Computer Press, 1997**

[2] **Android 2, Mark L. Murphy, Praha: Computer Press, 2011**

[3] **http://developer.android.com/index.html**

[4] **Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner: C# 2008, Programujeme profesionálně, Computer Press, duben 2009, ISBN: 978-80-251-2401-7**

[5] **M. MacDonald, M. Szpuszta, ASP.NET 3.5 a C# tvorba dynamických stránek profesionálně, Zoner press. Brno 2008**

Vedoucí diplomové práce: **Ing. Přemysl Svoboda**
Ústav mechatroniky a technické informatiky

Konzultant diplomové práce: **Ing. Pavel Tyl**
Ústav mechatroniky a technické informatiky

Datum zadání diplomové práce: **10. října 2013**

Termín odevzdání diplomové práce: **16. května 2014**

V. Kopecký

prof. Ing. Václav Kopecký, CSc.
děkan

Technická univerzita v Liberci - 14 - L.S.

Kolář

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci dne 10. října 2013

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

Podpis:

Poděkování

Na tomto místě bych rád poděkoval vedoucímu mé diplomové práce Ing. Přemyslu Svobodovi za jeho rady a poznatky, které byly velice užitečné a pomohly mi k realizaci mé práce. V neposlední řadě bych také rád poděkoval rodině za umožnění studia na vysoké škole a za jejich podporu při studiu.

Abstrakt

Tato práce by měla čtenáře seznámit s tématikou synchronizace databází mobilních zařízení se serverovou databází. Práce popisuje postupy a doporučení při vývoji knihoven pro operační systém Android. Dále čtenáře seznamuje s užitečnými komponentami operačního systému Android, které jsou vhodné pro řešení problematiky synchronizace databází. Jedná se o Služby a Systémová vysílání. Dále se zaměřuje na vývoj serverových aplikací na platformě .NET. Práce seznamuje čtenáře se základy vývojového prostředí Visual Studio 2013 a se založením nového projektu Web API. Dále práce popisuje základní strukturu projektu a jeho propojení se serverovou databází Microsoft SQL.

Praktická část práce popisuje návrh vlastní synchronizace databáze mobilního zařízení se serverovou databází. Práce popisuje vývoj klientské části v podobě knihovny pro operační systém Android a serverové části v jazyce ASP.NET. Dále práce popisuje, jak tento navržený a vytvořený balík použít. Na závěr práce popisuje testování synchronizace pomocí aplikace pro operační systém Android, do které byla implementována vytvořená knihovna. Pro lepší přehled o datech a pro kontrolu synchronizace byla vytvořena a v práci popsána webová aplikace. Ta umožňuje přihlášení uživatelů a zobrazení synchronizovaných dat, která jsou uložena v serverové databázi.

Klíčová slova

Synchronizace databází, OS Android, ASP.NET, Web API, Microsoft SQL

Abstract

This thesis introduces the reader to the topic of database synchronization between mobile devices and server database. It describes the procedures and recommendations for the development of libraries for Android operating system. Moreover there is an explanation of useful components of Android operating system, which are suitable for dealing with the synchronization of databases. It includes Services and Broadcasts. The thesis also focuses on the development of a server applications on .NET platform. It gives reader introduction to the basics of Visual Studio 2013 and creating of new project with Web API. It also describes the basic structure of the project and its integration with Microsoft SQL server database.

The practical part of the work describes design of proprietary synchronization between mobile and server database. Further it contains the implementation of a client application using library for Android operating system and server component in ASP.NET. The thesis also describes how to apply the designed and created package. Finally, the work introduces a test of synchronization using an application for Android operating system, in which this library was implemented. For a better overview of the data and synchronization control there has been created a web application as a part of this work. The web application allows users to log in and view synchronized data, which are stored in the server database.

Keywords

Database synchronization, OS Android, ASP.NET, Web API, Microsoft SQL

Obsah

Seznam obrázků

Seznam ukázek kódů

Seznam zkratek

OS – Operační systém

GSM – Globální Systém pro Mobilní komunikaci

ORM – Objektově relační mapování

MVC – Model, Pohled a Řadič

UUID – Univerzálně jedinečný identifikátor

XML – Rozšířitelný značkovací jazyk

GPS – Globální polohovací systém

UI – Uživatelské prostředí

VS – Visual Studio

API – Aplikační programové rozhraní

JSON – JavaScriptový objektový zápis

MS SQL – Microsoft SQL

HTTPS – Zabezpečený hypertextový transportní protokol

URL – Jednotný lokátor zdrojů

Úvod

Problematika synchronizace dat je obecně známa. Pro mobilní platformu však nelze nalézt mnoho existujících řešení. Původním záměrem bylo implementování synchronizace do již vytvořené aplikace, která byla realizována v rámci bakalářské práce [1] a následně v rámci diplomového projektu [2] rozšířena. Způsob implementace synchronizace byl zvolen formou knihovny pro Operační systém (OS) Android, vzhledem k tomu, že spousta mobilních aplikacích by synchronizaci mohla využít pro zálohu uživatelských dat.

Přistoupilo se tedy k návrhu a vývoji knihovny pro OS Android. V době psaní zadání se nepodařilo najít žádný podobný balíček, který by umožňoval synchronizaci mobilního zařízení se serverovou databází. V závěrečné fázi realizace této práce se objevila diplomová práce [3] na podobné téma. Řešení a použití je však naprosto odlišné. Volba operačního systému nebyla již možná vzhledem k plánům implementace do již existující aplikace. Navíc OS Android je stále nejrozšířenější.

Samotná knihovna by měla umožňovat automatickou synchronizaci lokální databáze se serverovou databází. Důraz je kladen na objem přenosu dat, protože mobilní zařízení často k připojení využívají Globální Systém pro Mobilní komunikaci (GSM) a dnešní datové balíčky limitují uživatele množstvím přenesených dat plnou podporovanou rychlostí. Dále byl kladen důraz na jednoduchost implementace a použití v konkrétní aplikaci. Bylo tedy nutné promyslet a zvážit způsob konfigurace knihovny. Knihovna musí nějakým způsobem umožňovat vytvoření a práci s lokální databází a také síťovou komunikaci. Po zvážení různých možností se přistoupilo k využití knihovny ActiveAndroid [4], která zajišťuje práci s lokální databází pomocí objektů a jejich instancí daného jazyka. Využívá se přitom objektově relačního mapování (ORM). Vytváření databáze a práce s ní je prováděna pomocí tříd a metod napsaných v jazyce Java.

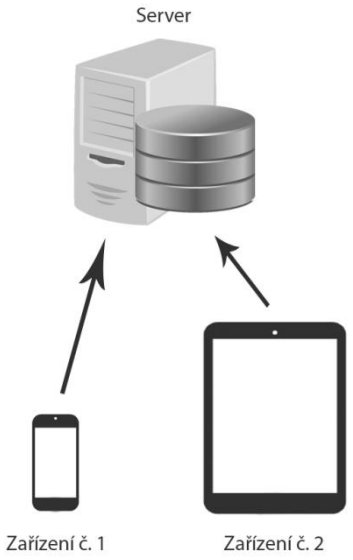
Nedílnou součástí celého synchronizačního balíčku je i serverová část. Pro její implementaci byla zvolena platforma .NET z důvodu dostupnosti serveru na této platformě. Vývoj serverové aplikace je založen na návrhovém vzoru Model, Pohled a Řadič (MVC), vytvořená implementace serverové části je tedy lehce přizpůsobitelná pro jinou aplikaci. S mírnými úpravami je snadné vytvořit další implementaci serverové části k použití knihovny v nové aplikaci.

1 Teoretická část

1.1 Synchronizace databází

Synchronizací databází se rozumí přenos záznamů do všech propojených databází, který se provádí automaticky bez nutnosti zásahu uživatele. Například aplikace pro zaznamenávání poznámek na mobilním telefonu využívá vestavěnou SQLite databázi. Dále existuje stejná desktopová verze, kde by poznámky měly být také dostupné. V tuto chvíli je potřeba poznámky mezi mobilní a desktopovou verzí synchronizovat. Většinou zde figuruje jako hlavní článek nějaký centrální server s databází, který řídí synchronizaci a uchovává všechna data. Všechny ostatní aplikace s vlastními databázemi se na tento centrální server dotazují a odesílají mu svá data. Tímto přístupem je zařízena synchronizace všech dat a všechny konkrétní programy mají data přístupná i offline bez využití centrálního serveru.

Během synchronizace mohou také nastávat různé konfliktní stavy, se kterými je nutné se vypořádat a zajistit, aby nedocházelo ke zbytečné ztrátě nebo k redundanci dat. Jedním takovým stavem může být úprava záznamu. Kdyby byl upraven daný záznam na zařízení č. 1, které není připojeno k internetu (tedy nedošlo by k synchronizaci s centrálním serverem), a následně by byl na zařízení č. 2 upraven a synchronizován na centrální server stejný záznam, tak by po připojení zařízení č. 1 k internetu došlo k přepsání starší verzí záznamu (viz Obrázek 1).



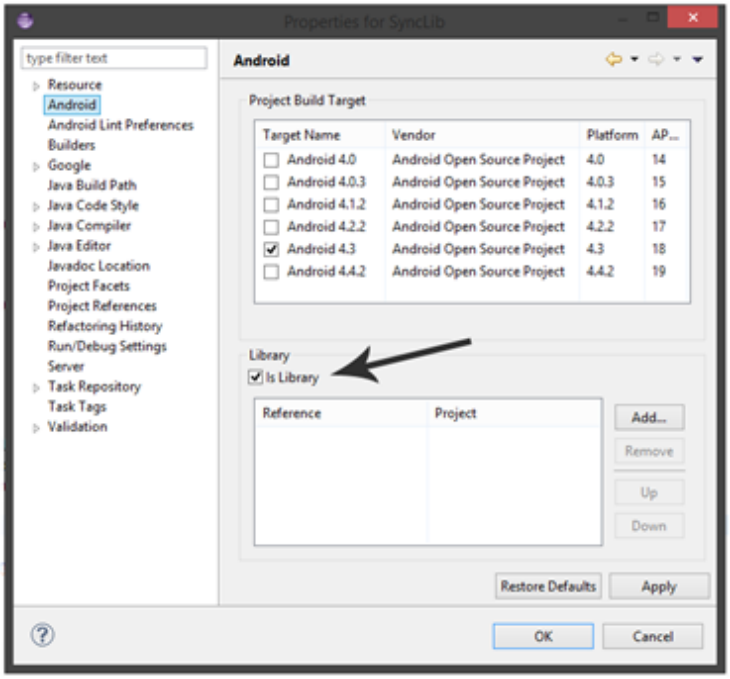
Obrázek 1: Schéma synchronizace

Tomuto je ve většině případů nutné předejít. Z tohoto důvodu se může ke každému záznamu přidat atribut s časovou informací o poslední úpravě. Tím se zaručí, že bude vždy možné identifikovat nejnovější záznam. Tento problém by bylo možné přenést z jednoho záznamu na jednotlivé atributy záznamu. Zde by bylo nutné vytvoření časových razítek ke každému atributu, což by bylo datově náročné. Dále by toto bylo možné provádět různým porovnáváním záznamů, což by však bylo časově náročné a vzhledem k přenosu pomocí internetu i datově náročné. Navíc není moc pravděpodobné, že by uživatel upravoval daný záznam na několika zařízeních zároveň, aniž by proběhla synchronizace s centrálním serverem.

Dalším problémem, který je nutné vyřešit, je jednoznačná identifikace záznamu. Většina databází využívá vlastní unikátní identifikátory v rámci jedné tabulky. Tyto identifikátory se většinou automaticky inkrementují, takže by nebylo vhodné jejich použití v rámci celé synchronizace. Nejlépe bude toto vysvětleno na příkladu. Na zařízení č. 1 bychom vytvořili dva záznamy, které by měly identifikátor roven 1 a 2, tyto záznamy by byly synchronizovány na centrální server s těmito identifikátory. Dále bychom vytvořili další dva záznamy na zařízení č. 2 bez předchozí synchronizace. Zde by vytvořené záznamy měly také identifikátory rovny 1 a 2. Kdybychom tyto záznamy synchronizovali, přepsaly by se původní záznamy na centrálním serveru a došlo by ke ztrátě dat. Tento problém je možný vyřešit použitím univerzálně jedinečného identifikátoru (UUID). Jedná se o unikátní identifikátor, který je vytvářen pomocí hašovacích funkcí (hash function). Existuje několik typů. UUID by měl zajistit, že každý záznam bude mít naprosto unikátní identifikátor, který se nebude opakovat ani na jiných zařízeních. Každý záznam tedy bude mít svůj lokální identifikátor, který bude v každé databázi jiný, a automaticky vygenerovaný a přiřazený UUID, který bude využit pro jednoznačnou identifikaci záznamu. UUID bude záznamu přiřazen pouze při jeho vytváření.

1.2 Knihovna pro OS Android

Knihovny pro OS Android se vytvářejí obdobným způsobem jako běžné aplikace. Ve vývojovém prostředí se založí nový projekt a pouze se v jeho nastavení zvolí možnost, že se jedná o knihovnu (viz Obrázek 2).



Obrázek 2: Nastavení projektu jako knihovny

Při vývoji aplikací pro OS Android je možné se setkat se dvěma druhy knihoven. Prvním druhem je klasický .jar soubor, který obsahuje všechny sestavené zdrojové kódy a jeho použití je jednoduché. Pro použití tohoto typu knihovny je nutné pouze nakopírovat daný .jar soubor do adresáře *libs* v konkrétním projektu. Tímto je knihovna ihned přístupná a je možné její použití. Druhým typem je knihovna s tzv. zdroji (resources). Mezi ně patří soubory napsané rozšiřitelným značkovacím jazykem (XML), soubory s rozvržením prvků (layout), dále soubory s hodnotami, textovými řetězci a další zdrojové soubory, které jsou nutné pro běh dané knihovny. Tento druh knihovny není možné sestavit do jednoho jediného souboru. Použití tohoto typu knihovny je složitější. Je nutný její import do pracovního adresáře (workspace), kde se nachází projekt, ve kterém má být použita knihovna. Dále je nutné její přidání v nastavení projektu. Následně je možné její využití jako v předešlém případě.

Je tedy dobré se vyvarovat použití knihovny zdrojů a použít pouze třídy jazyka Java, které je možné sestavit do .jar souboru. Vhodné je také vytvářet dokumentační řetězce minimálně ke všem veřejným metodám, protože vývojové prostředí je schopno tyto řetězce programátorovi zobrazit a ulehčit mu tak práci. Většina knihoven má plnit nějakou funkci, je tedy obvyklé vytvořit metodu nebo rozhraní pro její inicializaci a následně veřejné metody, které budou plnit funkci knihovny. Ostatní metody by měly být privátní, aby se zbytečně nezobrazovaly programátorovi, který bude knihovnu používat. Mohlo by tak docházet k nekorektní funkci dané knihovny. Veřejné by tedy

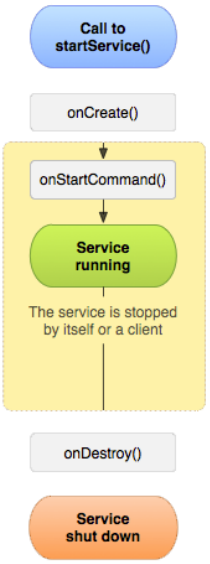
měly být opravdu jen ty metody, které jsou použitelné k dané funkcionalitě. Ne vždy je však toto možné. Je tedy dobré zachytit tuto skutečnost v dokumentaci.

Často také knihovny poskytují abstraktní třídy, které jsou v konkrétní aplikaci využívány. Důležité je použít klíčové slovo *final* u metod, které jsou určené pouze pro vykonávání funkce v knihovně. Tímto je zaručeno, že programátor, který bude využívat abstraktní třídu, nebude schopen měnit tělo těchto metod, což by mohlo vést k nekorektní funkci.

1.3 Služby

Služby (services) slouží k vykonávání úkonů na pozadí. Slouží například k přehrávání hudby, stahování velkých souborů, k periodickému kontrolování aktualizací apod. Služby na rozdíl od Aktivit (základní komponenta aplikace pro OS Android) nemají uživatelské prostředí, vykonávají pouze určitou činnost na pozadí. Služby náleží vždy nějaké aplikaci. Služba je schopna běžet kdykoliv. Aplikace poskytující službu nemusí být vůbec zapnutá. Proto je důležité hlídat ukončení služby, neboť by jinak mohla plýtvat prostředky. Je tedy možné během vykonávání služby pracovat s ostatními aplikacemi. Stejně jako aktivity mají i služby nějaké metody, které je povinné implementovat (viz Obrázek 3). Jedná se o:

- *onStartCommand*: Tato metoda slouží k inicializaci služby a je volána, když je služba spouštěna např. z aktivity. Pomocí této metody je také možné předat nějaká data z aktivity do služby.
- *onCreate*: Stejně jako u aktivit je tato metoda volána pouze při vytváření služby. Pokud je již služba spuštěna, tato metoda se nevolá.
- *onDestroy*: Tato metoda je volána, když služba skončí svoji činnost a je recyklována.



Obrázek 3: Životní cyklus služby [5]

Důležité je ošetřit ukončení služby. Je nevhodné, aby služba stále běžela, když už není potřeba. Nesprávným zacházením se službami může docházet k velkému úbytku kapacity baterie v důsledku využívání procesoru nebo například globálního polohovacího systému (GPS). Když služba dokončí vykonávanou práci, je nutné ji ukončit voláním metody *stopSelf*. Službu je také možné ukončit z jiné aktivity, a to voláním metody *stopService*. OS Android poskytuje dva druhy služeb.

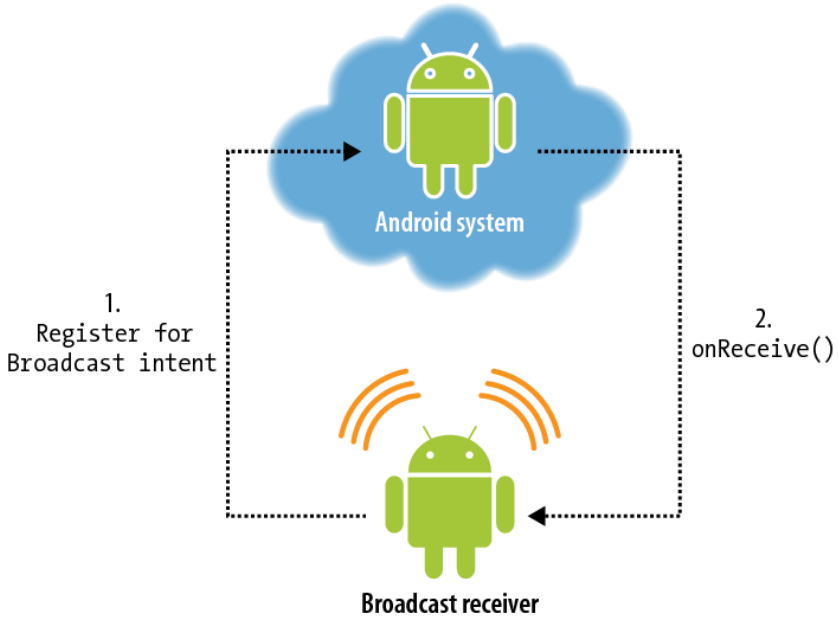
První jsou klasické běžné služby (třída Service), které běží ve stejném vláknu jako uživatelské prostředí (UI) aplikace. Pokud by tedy měla služba vykonávat nějaké složitější operace, je doporučováno vytvoření vlastního vlákna ve službě. Jinak by mohlo docházet k viditelnému zasekávání UI aplikace. Je také možné využití časovače (třída TimerTask) k periodickému spouštění nějaké akce. Tato třída umožňuje periodické spouštění určité části kódu. Každý úkol je spouštěn ve svém vlastním vláknu, takže není nutné ve službě vytvářet další vlákno.

Druhým typem je služba spouštěná na základě záměru (třída IntentService). Tato služba běží již automaticky ve svém vlastním vláknu. Tento typ služby funguje jako tzv. fronta. Při prvním volání metody *startService* je služba vytvořena a zavolána metoda *onHandleIntent*, ve které jsou službě předána a zpracována data. Pokud vykonávání skončí, je služba recyklována. Pokud však během vykonávání je znovu volána metoda *startService*, je tento požadavek zařazen do fronty. Po skončení vykonávání metody *onHandleIntent* je opakovaně volána metoda *onHandleIntent* s dalšími daty, která jsou

ve frontě. Když je fronta prázdná, služba se sama automaticky ukončí voláním metody *stopSelf*, takže programátor se o ukončení nemusí starat jako u předešlého typu služby.

1.4 Broadcast

Jedná se o vysílání informace o nějaké události. V OS Android existuje několik systémových volání, která jsou volána, když nastane nějaká situace. Například zařízení se připojí k internetu a operační systém vyšle zprávu o vytvořeném připojení.



Obrázek 4: Schéma funkce broadcast vysílání

Důležitou částí je také přijímač (reciever), který dokáže na tato volání reagovat. V dané aktivitě, kde je konkrétní volání z nějakého důvodu zajímavé, je nutné registrovat daný přijímač pro konkrétní volání. Toto by se mělo provádět v metodě *onResume* a v metodě *onPause* by se měl přijímač odhlásit. Toto je standardní postup, neboť kdyby aktivita nebyla zrovna aktivní, tak by docházelo k problémům, protože by registrovaný posluchač v případě daného vysílání začal vykonávat nějakou akci na neaktivní aktivitě. Jak bylo již zmíněno, v OS Android existuje několik systémových volání, je však možné vytvářet vlastní volání, která mohou také využívat i ostatní aplikace. Tato vlastní volání (broadcasty) je nutné zapsat do manifestu aplikace, aby systém měl informace o existujících voláních [6].

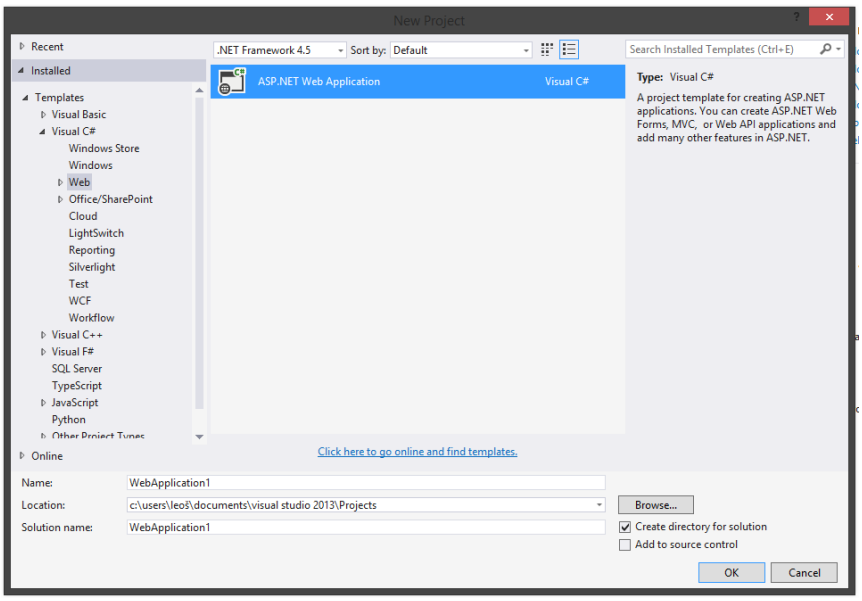
1.5 Web API ASP.NET server

ASP.NET je webová platforma pro tvorbu webových serverů od společnosti Microsoft. Tato platforma je postavena na .NET Framework, takže všechny jeho možnosti jsou zde také dostupné. Pro tuto platformu je možné programovat například v jazyce C# nebo Visual Basic. Pro vývoj webové aplikace je potřeba vývojové prostředí Visual Studio (VS).

Základem Web aplikačního programového rozhraní (API) je návrhový vzor MVC. Tento přístup je velice jednoduchý a odděluje od sebe jednotlivé části aplikace. První částí jsou modely (model). Jedná se o datové modely, které reprezentují uživatelská data. V tomto případě modely reprezentují tabulky databáze, kterou Web API využívá. Druhou částí jsou pohledy (view). Tedy uživatelská rozhraní dané aplikace. Poslední částí jsou řadiče (controller). Jedná se o vlastní řídící logiku aplikace. Všechny tyto části je možné vyvíjet nezávisle, protože mezi sebou mají jen minimální vazby.

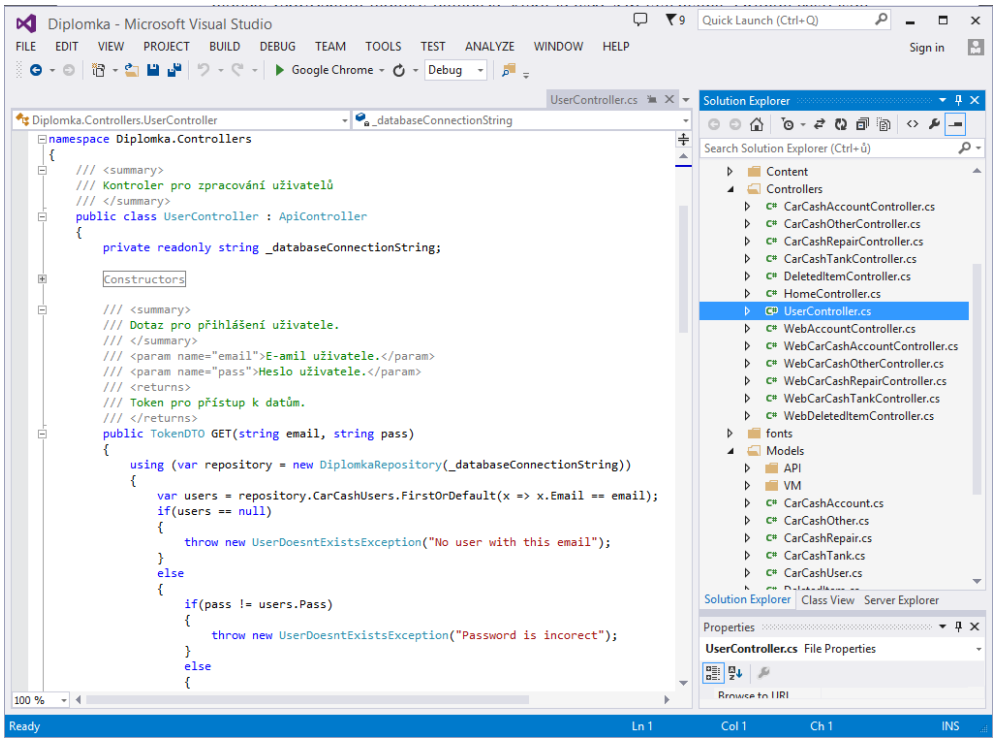
1.5.1 Vytvoření nového projektu

Pro vytvoření nového projektu potřebujeme VS, jak bylo zmíněno výše. Po instalaci a spuštění VS je možné založit nový projekt. Když je zvolena volba nový projekt, zobrazí se okno s výběrem typu projektu. Pro Web API je volba následující: Instalováno (Installed) −> Šablony (Templates) −> Visual C# −> Web a zde zvolit jedinou možnost ASP.NET Web Application [7]. (viz Obrázek 5)



Obrázek 5: Vytvoření nového projektu

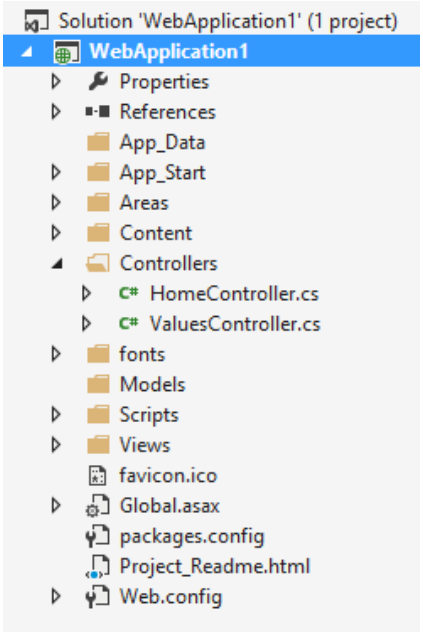
Dále je nutné vybrat umístění a pokračovat ve volbě. Zobrazí se další okno pro výběr druhu projektu. Zde se zvolí Web API a tato volba nám vytvoří ukázkovou webovou aplikaci, kterou je možné již spustit. VS (viz Obrázek 6) umožňuje spustit projekt na lokálním vývojovém serveru. Dokonce je možné ladění projektu, takže lze vše odladit ještě před nahráním aplikace na skutečný server.



Obrázek 6: Visual Studio 2013

1.5.2 Popis struktury projektu

Na obrázku je vidět základní struktura nového projektu (viz Obrázek 7). Tento typ projektu poskytuje webové API a webovou aplikaci v jazyce ASP. Důležitými složkami jsou *Controllers*, *Models*, *Views*. Tyto složky navazují na zmíněný návrhový vzor MVC.



Obrázek 7: Základní struktura projektu

Ve složce *Models* se tedy definují třídy, které reprezentují tabulky v databázi, a další pomocné třídy reprezentující například objekty, které poskytuje API v podobě JavaScriptového objektového zápisu (JSON).

Ve složce *Controllers* se nacházejí řadiče, které obsluhují všechny akce serveru. Pokud je vytvářen nový řadič, je k němu automaticky vygenerován i pohled ve složce *Views*. Pokud však je řadič určen k poskytování dat jako webové API, není nutné pohled nijak upravovat, protože není potřeba, data jsou poskytována v JSON podobě. Pohledy jsou potřeba pro webovou aplikaci, která je zobrazitelná v prohlížeči.

1.5.3 Propojení projektu s databází

Takto vytvořený projekt je jen aplikací pro daný server. Důležitou součástí Web API je databáze. Dobrou volbou je Microsoft SQL (MS SQL) databázový server od firmy Microsoft, ke kterému se lze z aplikace připojit. Konfigurace připojení aplikace k databázi se provádí v souboru Web.config přidáním následujícího řádku do sekce appSettings (viz Ukázka kódu 1).

```
<add key="název_kontextu" value="Server=adresa_serveru, port_serveru;
Database=název_databáze;User ID=jméno_uživatele;Password=heslo;
Trusted_Connection=False;Encrypt=True;MultipleActiveResultSets=True;" />
```

Ukázka kódu 1: Konfigurační řetězec pro propojení s databází

Tento příkaz je nutné upravit dle konkrétní implementace, je nutné nastavit adresu databázového serveru s portem, jméno databáze, uživatelské jméno a heslo pro přístup k databázi. Dále je nutné nainstalovat do projektu balíček pro práci s databází. Balíčky je možné vyhledávat na oficiálních stránkách [8], kde jsou dostupné různé balíčky (obdoba knihoven). Pro komunikaci aplikace s databází je potřeba balíček s názvem EntityFramework, který se nainstaluje do projektu zadáním: „Install-Package EntityFramework“ do *PackageManagerConsole*, která je dostupná ve vývojovém prostředí. Tímto se nainstaluje vybraný balíček a je možné pokračovat v implementaci propojení s databází.

Dalším krokem je vytvoření složky *Repositories*, kde je nutné vytvořit několik tříd, které jsou přiloženy k této práci na CD. V těchto třídách se provede konfigurace propojení a následně je možné vytvářet pouze modely, které reprezentují tabulky v databázi. Důležité je dodržet stejnou strukturu. Díky této konfiguraci a využití EntityFramework není nutné napsat jediný SQL příkaz. Při implementaci se pracuje pouze s objekty a o zbytek se postará již zmíněný EntityFramework. Podrobnější postup implementace bude popsán v praktické části, protože souvisí s použitím navrženého řešení.

1.5.4 JSON a XML reprezentace objektů

Oba formáty jsou využity pro jednoznačnou reprezentaci objektů pro přenos pomocí internetu. Oba formáty jsou multiplatformní a nijak nemění nesená data. JSON je odlehčenou verzí XML, kde až 40% obsahu tvoří vlastní značky, které nenesou data. Pro účely webového API je tedy lepší využít JSON vzhledem k menší datové náročnosti.

JSON umožňuje reprezentaci jednoduchých objektů, polí a komplexnějších objektů, které obsahují například jiné objekty nebo pole jiných objektů. Velice snadno se v něm člověk orientuje pouze v textové podobě a je jednoduchý pro rozklíčování v klientské aplikaci. Existují různé knihovny, které toto ještě ulehčují. [9]

Webové API popsané dříve poskytuje primárně data v XML podobě. Pro použití JSON reprezentace je nutné opět doinstalovat do projektu další balíček. Balíček pro podporu JSON se jmenuje Nancy Serialization JSONNet. Jeho instalace se provede zadáním příkazu: „Install-Package Nancy.Serialization.JsonNet“ do *PackageManagerConsole* ve vývojovém prostředí. V ukázce je pak zobrazena

implementace objektu tak, aby využil nainstalovaný balíček a byl reprezentován pomocí JSON objektu (viz Ukázka kódu 2).

```
[DataContract]
public class ErrorDTO
{
    [DataMember(Name = "id")]
    public int Type { get; set; }

    [DataMember(Name = "message")]
    public string Message { get; set;
}
```

Ukázka kódu 2: Definice objektu pro JSON reprezentaci

```
{
    "error":
    {
        "id": 110,
        "message": "The request is invalid."
    }
}
```

Ukázka kódu 3: JSON reprezentace daného objektu s konkrétními daty

2 Praktická část

2.1 Návrh synchronizace

Návrh synchronizace vychází z rešerše zpracované v teoretické části této práce. V návrhu je snaha ošetřit všechny možné chybové stavy, aby byla synchronizace co nejefektivnější a nedocházelo ke ztrátám informací. Důraz je kladen na minimální datovou náročnost. Z tohoto důvodu je synchronizace založena na konkrétních samostatných záznamech. Nedochází ke zbytečnému přenosu již odeslaných záznamů. Také by mělo být umožněno pracovat s lokální databází v offline módu. Po připojení k internetu by se všechny provedené změny měly promítnout do centrální databáze. Z tohoto důvodu bude nutné si dát pozor na pořadí prováděných úkonů v průběhu synchronizace.

Byla zvolena architektura s centrálním serverem a serverovou databází, což bude zajišťovat stálé uložení záznamů. Server bude záznamy poskytovat klientským aplikacím a také je přijímat z klientských aplikací. Dále bude poskytovat webovou aplikaci pro zobrazení dat ze serverové databáze. Data budou zobrazována pouze přihlášenému uživateli. Budou mu zobrazována pouze jeho vlastní data. Jako technologie pro implementaci serveru byla zvolena platforma od společnosti Microsoft, konkrétně framework .NET s programovacím jazykem ASP. Jako databázový server byl zvolen MS SQL, také od společnosti Microsoft. Klientské aplikace budou tvořeny knihovnou pro OS Android, kterou je nutné implementovat v konkrétní aplikaci. Knihovna bude tvořena tak, aby bylo jednoduché i použití jiné technologie na straně serveru, kde bude muset být dodržena struktura poskytovaných metod popsaná v příloze této práce.

2.1.1 Uživatelé

Prvním krokem k provozu synchronizace je identifikace uživatelů. Navržený systém by tedy měl umožňovat registraci a přihlášení uživatelů. Na serverové části bude nutné implementovat pro toto metody a v centrální databázi vytvořit tabulku pro uchovávání informací o uživatelích. Klientská část by pak měla poskytovat metody pro jednoduchou implementaci registrace a přihlášení v konkrétní aplikaci. Povinnými údaji by měl být e-mail (pro jednoznačnou identifikaci uživatele) a heslo v podobě nějaké hašovací funkce (pro zabezpečení). Při registraci by měla serverová část přidělit

uživateli nějaký jednoznačný identifikátor, který bude využíván pro identifikaci uživatele při volání dalších metod.

2.1.2 Operace se záznamy v knihovně

- Nový záznam: Vytvoření nového záznamu by nemělo představovat žádný problém při synchronizaci. Jediné, co je potřeba ošetřit, je přidělení unikátního identifikátoru vzhledem k problému popsanému v teoretické části práce. V navrženém systému bude využíván UUID, který bude generován pouze při vytváření nového záznamu v klientské aplikaci. Jeho generování bude probíhat automaticky v knihovně a programátor implementující knihovnu se o toto nebude muset starat. Dále bude záznamu automaticky přidán identifikátor, který bude určovat, zda byl již záznam odeslán na server. Takto bude záznam uložen do lokální databáze a bude proveden pokus o odeslání záznamu na server. V případě úspěšného odeslání se zmíněný identifikátor změní na status odesláno. Tím je synchronizace dokončena. Pokud by se záznam nepodařilo odeslat na server, identifikátor by zůstal ve stavu neodesláno a pokus o odeslání záznamu by byl proveden při volání metody pro kompletní synchronizaci, která bude popsána dále.
- Úprava záznamu: Zde již může docházet ke ztrátě dat, jak bylo popsáno v teoretické části práce. Z tohoto důvodu bude knihovna automaticky přidávat ke každému záznamu informaci o času poslední úpravy záznamu. Tento údaj bude společně s vlastními daty záznamu přenášen do centrální databáze, kde bude také uchováván. Při příjmu záznamu do centrální databáze proběhne kontrola, zda již existuje záznam s daným UUID. Pokud takový záznam existuje, porovnají se časy editace a zachová se pouze novější záznam. Odesílání upraveného záznamu funguje stejně jako vytvoření nového záznamu. Rozdílem je, že se negeneruje nový UUID, ale dojde pouze ke změně času úpravy záznamu, k nastavení identifikátoru o odeslání na stav neodesláno a k provedení pokusu o odeslání.
- Smazání záznamu: Vzhledem k požadované funkcionalitě v offline modu bude knihovna automaticky přidávat tabulku se smazanými záznamy. Toto řešení je zvoleno, protože se tak umožní získávat data

z uživatelských tabulek bez filtrovacích dotazů. Při smazání záznamu se záznam ihned z dané tabulky smaže a uloží se UUID záznamu a jméno tabulky smazaného záznamu do této automaticky vytvořené tabulky. Následně je proveden pokus o odeslání záznamu na server. Pokud je záznam úspěšně odeslán, je v lokální klientské databázi smazán, protože zde již není potřeba. Pokud záznam není v pořádku odeslán, je ponechán v této tabulce a následně při volání metody pro kompletní synchronizaci se knihovna pokouší odeslat všechny záznamy i z této tabulky.

2.1.3 Zpracování záznamů na straně serveru

- Nový záznam/upravený záznam: Metoda zpracovávající tyto dvě operace je společná. Nejprve se porovná UUID přijatého záznamu s centrální databází. Pokud se v centrální databázi nachází záznam s tímto UUID, dochází k úpravě záznamu. Pokud takový záznam neexistuje, dochází k vytvoření nového záznamu. Všechna data jsou předána z klientské aplikace. Serverová aplikace pouze přidá další atribut s informací o čase nahrání na server. Ten je nutné uchovávat pro metodu, která poskytuje data pro klientskou aplikaci. Dále server přidá vlastní lokální identifikátor.
- Smazaný záznam: Přijatý smazaný záznam je uložen do speciální tabulky smazaných záznamů, stejně jako v knihovně. Zde je nutné však záznamy uchovávat trvale. Není totiž možné jednoduše zajistit smazání záznamu, když se promítne jeho smazání na všechna přihlášená zařízení. Vzhledem k serverové centrální databázi to nijak nevadí. Pouze se budou uchovávat do jisté míry již nepotřebná data. Vzhledem ke kapacitě serverových disků a objemu smazaných dat je tento návrh vyhovující.

2.1.4 Poskytování záznamů

Serverová část musí také poskytovat metody pro získávání nových záznamů a smazaných záznamů. Aby docházelo k přenosu pouze nejmenšího možného množství dat, je zaveden již zmíněný atribut o informaci času uložení do centrální databáze. Knihovna si pro každou tabulku uchovává čas posledního úspěšného stažení dat. Tento čas je při dalším volání odesílán na server, který toto vyhodnotí a vrátí klientské aplikaci pouze záznamy, které byly nahrány po poslední aktualizaci. Ve stejném dotazu

server vrací nově smazané záznamy od posledního stažení. Díky tomuto je docíleno minimálního přenosu dat. Pro jistotu je porovnávání časů redukováno na dny a ne na minuty či sekundy. Bylo by tedy možné ještě datovou náročnost mírně zredukovat, ale vzhledem k návrhu synchronizace by úspora nebyla nijak velká.

Knihovna na straně klienta automaticky provádí ukládání času posledního úspěšného stažení záznamů konkrétní tabulky. Takže i tímto je redukován přenos dat. Každá tabulka má vlastní dotaz a vlastní hodnotu pro poslední aktualizaci. Aktualizace sice probíhá najednou pro všechny tabulky, ale pomocí několika dotazů. Proto je zaveden parametr s informací o času aktualizace pro jednotlivé tabulky, aby se předešlo problémům.

2.2 Implementace knihovny pro OS Android

Nejprve bylo nutné navrhnout způsob konfigurace knihovny při použití v nějaké aplikaci. Knihovna bude vytvářet lokální databázi v aplikaci, takže je nutné nějakým způsobem umožnit návrh databáze v knihovně. Cílem tedy bylo navrhnout knihovnu tak, aby bylo její použití naprosto jednoduché a aby dala programátorovi volnost pro definování konkrétní databáze.

První zvažovanou možností byl nějaký konfigurační soubor, například ve formátu XML. Programátor by tedy napsal konfigurační soubor, který by knihovna zpracovala a dle něho vytvořila databázi, kterou by následně obsluhovala. Takový přístup by asi nebyl moc efektivní. Musela by se navrhnout vhodná struktura konfiguračního souboru, která by musela být striktně dodržena. Samotná implementace knihovny by byla asi také velice problematická. V OS Android je práce s databází poněkud komplikovaná. Je nutné psát samotné SQL příkazy, které jsou vykonávány pomocí poskytovaných metod. Muselo by se tedy vyřešit nějaké zpracování konfiguračního souboru, ze kterého by se vygenerovaly příslušné SQL příkazy.

Od tohoto návrhu bylo ustoupeno, protože jsem již nějakou dobu využíval volně dostupnou knihovnu ActiveAndroid, která značně ulehčovala práci s lokální databází. Snaha tedy byla o využití této knihovny, protože její použití a definování struktury databáze je naprosto jednoduché. Proběhlo tedy zkoumání a nastudování této knihovny. Její zdrojové kódy jsou volně dostupné v systému správy verzí společnosti GitHub [10].

Bylo zjištěno, že knihovna je postavena na ORM. Jedná se o přístup, kdy jsou objekty v objektově orientovaném jazyce převáděny pomocí dostupných metod (např. reflexe) na relační databázové záznamy. Výsledkem je efektivní práce s databází pouze pomocí metody a příkazů daného jazyka bez nutnosti napsat jediný SQL příkaz. Lze si to představit tak, že nadefinovaná třída reprezentuje tabulku v databázi. Proměnné v dané třídě tedy reprezentují atributy tabulky.

2.2.1 ActiveAndroid

Jak již bylo zmíněno, jedná se o ORM knihovnu pro operační systém Android. Její použití je intuitivní. Knihovna neobsahuje žádné další soubory (resources), takže je zkompilovaná do jednoho .jar souboru. Tento soubor tedy stačí přiložit do nové aplikace a dle návodu uvedeného na stránkách knihovny použít. Použití zde nebude vysvětleno, protože ve výsledné knihovně bude trochu poupravené a bude uvedeno

v kapitole 2.4, která se bude věnovat použití nově vytvořené knihovny, pro kterou tato knihovna tvoří základ.

2.2.2 ActiveAndroid struktura projektu

V základním balíčku se nachází základní třída ActiveAndroid.java, kde se nacházejí inicializační statické metody. Není tedy nutné vytvářet instanci objektu. Stačí volat konkrétní metodu. Další důležitou třídou je Cache.java. Knihovna využívá vlastní mezipaměť (cache) pro uchovávání referencí, například pro již načtené záznamy z databáze. Pokud je požadován znovu nějaký záznam, je nejprve kontrolována mezipaměť, teprve poté se přistupuje k databázi. Tímto dochází k urychlení při práci se stejnými daty během jednoho spuštění aplikace. Jsou zde také uchovávány reference na objekty, které přímo pracují s databází.

K práci s databází slouží třída DatabaseHelper.java. Třída TableInfo.java slouží k uchovávání struktury tabulek. Pro každou tabulku je vytvořena instance této třídy a jsou v ní uchovány jednotlivé atributy. Této třídy se využívá při práci s databází a při převodu objektu na záznam do relační databáze.

Hlavní třídou, se kterou se setká i implementující programátor, je Model.java. Jedná se o abstraktní třídu, kterou je při implementaci nutné dědit. Tím je dědící třída zahrnuta do struktury databáze a reprezentuje jednu tabulku. Tato třída implementuje metody pro práci s konkrétními záznamy. Obsahuje metodu *save*, která provádí uložení záznamu do databáze, a metodu *delete*, která slouží ke smazání záznamu z databáze. Dále obsahuje další pomocné metody, například metodu pro převod získaného záznamu v podobě instance třídy Cursor (základní datová struktura v OS Android při práci s databází) do konkrétní instance třídy, která dědí ze zmíněné třídy Model.java. S touto třídou souvisí třída ModelInfo.java. Tato třída poskytuje metody, které při inicializaci knihovny mapují třídy vytvářeného projektu a zjišťují, které třídy dědí od Model.java. Tímto je zmapována struktura databáze a na základě toho je databáze vytvořena. Toto je prováděno pouze při prvním spuštění aplikace nebo při aktualizaci databáze. K tomu dochází, když se zvýší v konfiguračním řetězci verze databáze. Tento řetězec se nachází v manifestu aplikace a bude popsán dále.

Dalším balíčkem je *annotation*. Zde se nacházejí dvě třídy. Table.java, kde je definována vlastní anotace pro zadání názvu tabulky. Tato anotace se následně uvádí ve třídě, která dědí z Model.java. Nad názvem dané třídy se uvede tato anotace a jako parametr *name* se uvede název tabulky. Druhou třídou v tomto balíčku je Column.java,

kde je definována opět vlastní anotace pro definování atributů tabulky. Je zde opět parametr *name*, dále pak *notNull*, *unique*, atd. Tato anotace se pak uvádí nad jednotlivými proměnnými v dané třídě. Povinné je uvádět pouze parametr *name*, další jsou volitelné. Parametr pro datový typ se zde nenachází, protože ho knihovna určí z datového typu dané proměnné (viz Ukázka kódu 4).

```
@Table(name="CarCashOther")
public class CarCashOther extends Model
{
	@Column(name="AccountId", notNull=true, length=5,unique=true)
	private String mAccountId;
	@Column(name="Date")
	private String mDate;
	@Column(name="Km")
	private int mKm;
	@Column(name="Name")
	private String mName;
	@Column(name="Text")
	private String mText;
	@Column(name="Cash")
	private float mCash;
	@Column(name="CashType")
	private String mCashType;
}
```

Ukázka kódu 4: Definice tabulky databáze pomocí ORM

Dalším důležitým balíčkem je *query*. Zde se nachází několik tříd. Ty obsahují funkce, pomocí kterých jsou mapovány příkazy SQL jazyka. Jedná se o třídy Select.java, Delete.java, From.java, Update.java, Join.java, atd. Pomocí těchto tříd a jejich metod je možné získávat záznamy z databáze, mazat je a upravovat. Dokonce umožňují kladení podmínek, z jaké tabulky mají záznamy být a také například jaké mají mít atributy. Proto je práce s databází velmi jednoduchá a není nutné psát SQL příkazy, vše je na mapováno na objekty a jejich metody. Níže jsou pro lepší představu uvedené nějaké příklady použití těchto tříd a metod.

```
new Delete().from(Item.class).where("Id = ?", 1).execute();
```

Ukázka kódu 5: Smazání záznamu z tabulky Item s Id = 1

```
List<Item> = new Select().from(Item.class).where("Category = ?", 2)
            .orderBy("Name ASC").execute();
```

Ukázka kódu 6: Získání záznamů z tabulky Item, které mají Category = 2, a jsou seřazeny dle atributu Name

Dalším balíčkem je *serializer*. Zde se nacházejí třídy, které poskytují metody pro serializaci a deserializaci nestandardních datových typů, které není možné jednoduše uložit do SQLite databáze. Jedná se například o DateTime, BigDecimal, Calendar, atd.

Dalším důležitým balíčkem je *util*. Zde jsou uspořádány třídy, které zajišťují různé pomocné nástroje pro funkci knihovny. Nachází se zde třída pro nastavení logování v knihovně, které je možné konfigurovat v inicializační metodě. Dále je zde třída pro pomocné funkce mapování modelových tříd pomocí reflexe a také pomocné metody pro práci s SQLite databází.

2.2.3 Vlastní implementace knihovny

Výše popsaná knihovna posloužila jako základ nově vytvářené knihovny pro automatickou synchronizaci. Většina tříd ActiveAndroid knihovny byla ponechána bez úprav. Pouze jedna třída byla rozšířena pro správnou funkčnost. Jedná se o Model.java. Dále byly také přidány nové třídy, které byly nutné pro úplnou funkčnost. Dále budou popsány všechny úpravy a nové třídy knihovny.

Nová knihovna nese název SyncLib. Byl proto založen další balíček pro vlastní třídy, které nebyly součástí ActiveAndroid. V knihovně se tedy nachází dva základní balíčky. Pro ActiveAndroid je to com.activeandroid. Pro vlastní knihovnu je to cz.dostalleos.synclib. Toto rozdělení napomáhá lepšímu přehledu a odděluje nově tvořené části knihovny. Jak bylo zmíněno, v nově vytvářené knihovně došlo k úpravě jedné původní třídy. Tato třída zůstala v původním balíčku.

Jedná se o třídu Model.java. Tato třída reprezentuje rodičovskou třídu pro odvozené třídy, které pak reprezentují jednotlivé tabulky v databázi. Z důvodu synchronizace a jejího návrhu bylo nutné uchovávat nějaké povinné atributy. V původní verzi byl povinným atributem pouze identifikátor (*id*). Nyní jsou povinné atributy rozšířeny na *id, UUID, uploaded, updateDate*. Tyto atributy jsou implementovány naprosto stejně jako další volitelné parametry v děděných třídách. Tedy pomocí anotace. Všechny tyto parametry jsou definovány jako privátní, protože jsou generovány automaticky knihovnou a uživatel je nesmí nijak měnit, protože by tato skutečnost ovlivňovala chod synchronizace. Dále byl z důvodu přidání těchto parametrů upraven konstruktor, kde dochází pouze k prvotní inicializaci těchto proměnných. Dále byly přidány pomocné proměnné pro řízení synchronizace. S těmito proměnnými byly přidány metody pro nastavení a čtení jejich hodnoty, aby bylo dodrženo zapouzdření objektu. Všechny tyto metody byly definovány jako *final*, aby je vývojář nemohl měnit.

Bohužel bylo nutné tyto metody nastavit jako veřejné, protože k nim bylo potřeba přistupovat v rámci knihovny z jiných balíčků. U těchto metod je v dokumentačních řetězcích uvedeno, že by je vývojář neměl používat z důvodu zachování funkčnosti knihovny. První upravenou metodou je *delete* (viz Ukázka kódu 7). Původně tato metoda pouze smazala daný záznam v lokální databázi. Tato metoda byla přejmenována na *deleteFromDb*. V původní metodě se nejprve provede vytvoření nového záznamu v tabulce smazaných záznamů (dle návrhu). Potřebné údaje jsou získány z konkrétní instance modelové třídy. Následně je volána přejmenovaná metoda pro smazání záznamu z tabulky. Vytvoření záznamu v tabulce smazaných záznamů se potvrzuje voláním metody *save* na daném objektu. Tím je zajištěna synchronizace daného záznamu (toto bude popsáno dále).

```
public final void delete()
{
    DeletedItem item = new DeletedItem();
    item.setItemUUID(mUUID);
    item.setItemTable(mTableInfo.getTableName());
    item.save();
    deleteFromDb();
}
```

Ukázka kódu 7: Metoda delete

Další metodou je zmíněná metoda *save*. Tato metoda byla již v původní knihovně, nově je však zásadně rozšířená. V původní metodě probíhalo mapování instance objektu do záznamu v relační databázi. Nyní je přidán převod instance na JSON reprezentaci. Ta slouží k přenosu záznamu do centrální databáze pomocí zabezpečeného hypertextového transportního protokolu (HTTPS). Nejprve se však provádí několik operací. Pokud se jedná o nový záznam, je vygenerován unikátní identifikátor (*UUID*) [11], nastaven status o odeslání na hodnotu neodesláno a zaznamenán čas vytvoření. Pokud se jedná o již existující záznam, změní se pouze status o odeslání záznamu na neodesláno a dojde k úpravě času vytvoření neboli času úpravy záznamu. Dále se zde provádějí další operace, které slouží pouze pro vnitřní potřebu knihovny a ošetřují správnou funkčnost. Jedná se například o rozhodování, zda se jedná o akci, kterou vyvolal vývojář, nebo o akci, kterou vyvolala knihovna. Jelikož metoda *save* provádí jak uložení do lokální databáze, tak odeslání na server, je nutné ošetřit, aby bylo možné upravit záznam jen v lokální databázi a neprovedlo se odeslání na server. Toto je potřeba například při úspěšném odeslání záznamu na server. V této

chvíli je nutné upravit status záznamu na hodnotu odesláno. Toto však již není nutné odesílat na server.

Na konci metody tedy dochází k vlastnímu uložení do lokální databáze a k pokusu o odeslání záznamu na server. Toto je realizováno pomocí třídy IntentService [12], která bude popsána následně. Dále je zde nová metoda *getJson*, která vrací JSON reprezentaci dané instance objektu. Toto je potřeba při synchronizaci dříve neodeslaných záznamů. Místo volání metody *save* se volá tato metoda, protože záznam není nutné znovu ukládat do lokální databáze, ale pouze odeslat. Dále jsou zde dvě metody pro naplnění dané instance hodnotami z instance třídy Cursor nebo JSON objektu (viz Ukázka kódu 8). Tyto metody jsou potřeba například při načítání z lokální databáze nebo při příjmu nového záznamu ze serveru. Dále zde jsou pomocné metody.

```
if (value == null)
{
}
else if (fieldType.equals(Byte.class) || fieldType.equals(byte.class))
{
     sb.append("\"" + fieldName + "\":");
     sb.append((Byte) value);
     sb.append(",");
}
else if (fieldType.equals(Short.class) || fieldType.equals(short.class))
{
     sb.append("\"" + fieldName + "\":");
     sb.append((Short) value);
     sb.append(",");
}
.
.
.
sb.deleteCharAt(sb.length()-1);
sb.append("}");
sb.toString();
```

Ukázka kódu 8: Část kódu pro převod instance objektu do JSON reprezentace

Dále budou popsány třídy, které se nacházejí ve vlastním balíčku cz.dostalleos.synclib. V tomto kořenovém balíčku jsou zanořené další balíčky, kde jsou oddělené typové třídy.

První třídou je SyncLib.java. Tato třída se nachází přímo v kořenovém balíčku a jedná se o hlavní třídu celé knihovny. Tato třída poskytuje hlavní funkcionalitu knihovny. Vývojář implementující tuto knihovnu pracuje pouze s touto třídou a s vytvořenými modely, které dědí ze zmíněné modelové třídy Model.java. Tato třída je postavena na návrhovém vzoru Singleton. Může tedy existovat pouze jedna její

instance. Pro získání instance je zde metoda *getInstance* (viz Ukázka kódu 9). Všechna volání funkcí knihovny v této třídě je nutné mít v následujícím tvaru: *SyncLib.getIntsance().názevMetody()*.

```
public static SyncLib getInstance()
{
    if(mInstance==null)
    {
        mInstance = new SyncLib();
    }
    return mInstance;
}
```

Ukázka kódu 9: Singleton implementace třídy SyncLib.java

Hlavní metodou je *initializeSyncLib*, která provádí inicializaci knihovny při spouštění aplikace. Tato metoda přebírá tři parametry. Prvním je aplikační kontext aplikace. Druhým je textový řetězec s jednotným lokátorem zdrojů (URL), kde se nachází serverová část této knihovny. Posledním parametrem je booleovská hodnota, která umožňuje nastavení logování knihovny. Tato metoda zaručuje uchování těchto tří zmíněných parametrů po dobu běhu aplikace. Dále se zde provádí inicializace knihovny ActiveAndroid a registrace dvou přijímačů vysílání (třída BroadcastReciever), které slouží pro zachytávání zpráv o dokončení jednotlivých částí synchronizace. Tato metoda by měla být volána pouze jednou, a to při startu aplikace. Další metodou je *disposeSyncLib*, která provádí uvolnění zdrojů knihovny ActiveAndroid a ruší registraci dvou zmíněných přijímačů vysílání. Tato metoda by měla být volána pouze při ukončování aplikace. Dále jsou v této třídě implementovány metody pro registraci a přihlášení uživatelů. Metoda pro registraci je přetížená a je jí možné předat různé parametry. V prvním případě se předává e-mail, heslo a posluchač (listener). V druhém případě se předává e-mail, heslo, json a posluchač. Posluchač je vlastní třída knihovny, kterou je nutné předat těmto metodám. Slouží pro informování aplikace o dokončení registrace, která probíhá asynchronně. Parametr json může obsahovat další údaje uživatele ve formátu JSON a knihovna je automaticky přidá do objektu odesílaného na server. Toto je nutné zohlednit v konkrétní implementaci serveru. Metoda pro přihlášení funguje obdobně jako první metoda pro registraci. Obsahuje e-mail, heslo a posluchač (opět vlastní, ale jiný než předchozí). Další metoda provádí odhlášení uživatele, které je pouze lokální. Při odhlášení dochází ke smazání lokální databáze. Toto je z důvodu konzistence dat. Další metoda provádí kompletní synchronizaci databáze. Jedná se

o *syncDb*. Této metodě je nutné pouze předat posluchač, který bude informovat aplikaci o ukončení synchronizace. V této metodě je volána privátní metoda *uploadAllItems*. Tato metoda provádí nahrání všech neodeslaných záznamů. Odesílání záznamů je řešeno pomocí třídy IntentService. V této službě jsou postupně odeslány všechny neodeslané záznamy ze všech tabulek lokální databáze, včetně automaticky vytvořené tabulky *DeletedItems*. Jednotlivé záznamy pro každou tabulku jsou odeslány hromadně v jednom HTTPS volání. Pokud nejsou žádné neodeslané záznamy nebo je dokončeno odeslání záznamů, knihovna pokračuje k další metodě. O dokončení odesílání je knihovna informována při ukončování služby pomocí broadcast vysílání. Další metodou synchronizace je *getNewItems*. Zde probíhá opět spouštění služby, která postupně zkontroluje pro všechny tabulky nové záznamy a nově smazané záznamy na serveru. Při startu služby jí je předán název tabulky a čas její poslední aktualizace. V této metodě také probíhá kontrola, zda je uživatel přihlášen. Pokud není, tak synchronizace neproběhne (viz Ukázka kódu 10).

```
private final void startIntentGetService(String date, String tableName)
{
   SharedPreferences prefs = SyncLib.getInstance().getAppContext()
                             .getSharedPreferences(SyncLib.PREFS_NAME, 0);
   String token = prefs.getString(PREFS_TOKEN, null);
   if(token != null)
   {
       Intent intent = new Intent(SyncLib.getInstance()
                       .getAppContext(), GetItemIntentService.class);
       intent.putExtra(GetItemIntentService.EXTRA_UPDATE_DATE, date);
       intent.putExtra(GetItemIntentService.EXTRA_TABLE_NAME, tableName);
       SyncLib.getInstance().getAppContext().startService(intent);
   }
}
```

Ukázka kódu 10: Metoda pro zapnutí služby

Pro každou tabulku si knihovna uchovává čas poslední synchronizace, který je zde odesílán. Po ukončení této služby je opět vyslán broadcast, který zachytí knihovna ve své metodě. Zde pak proběhne volání předaného posluchače a další broadcast vysílání, pro které je možné si registrovat přijímač v aplikaci. Toto slouží k informování aplikace o dokončené synchronizaci. Tyto dvě zmíněné služby obstarají celou synchronizaci. Při vykonávání synchronizace jsou také ošetřeny všechny problémové stavy jako neaktivní připojení k internetu a nepřihlášený uživatel. Dále jsou zde další pomocné metody jako například posluchače pro asynchronní vykonávání HTTPS komunikace.

Pro síťovou komunikaci je zde použit balíček tříd, které umožňují provádět volání asynchronně a synchronně. V balíčku je využito obou typů HTTPS komunikace. Při registraci a přihlášení uživatele je využíváno asynchronního volání, protože knihovna běží v hlavním vláknu aplikace. Synchronní volání je využito ve službách odesílajících a přijímajících data. Tyto služby totiž běží ve vlastním vláknu a je tedy možné čekat na vyřízení komunikace přímo v tomto vláknu. Navíc je nutnost použít synchronního volání ve službách, které dědí z třídy IntentService. Protože jsou vykonávány jako fronta, tak by nebylo možné korektně reagovat na asynchronní přenos.

Dále byly v knihovně definovány určité entity. První se nazývá DeletedItem a slouží k automatickému vytvoření této tabulky v lokální databázi. Další entity jsou Error a ErrorHandler. Tyto dvě entity slouží syntaktické analýze (parseru) při síťové komunikaci a jsou použity v případech, kdy server vrátí nějaký druh chyby. V dalších balíčcích jsou implementovány zmíněné posluchače, služby a další pomocné třídy, například pro vytváření MD5 haše uživatelova hesla.

2.3 Implementace serverové části

Pro implementaci serverové části byla zvolena platforma .NET z důvodu možnosti zřízení serveru na této platformě. Vývoj pro tuto platformu je také velmi přívětivý, protože Webové API na této platformě je založeno na návrhovém vzoru MVC. Stejně tak vývoj webové aplikace je postaven na návrhovém vzoru MVC a je možné obě tyto služby implementovat v jednom projektu. Z tohoto důvodu je také vytvořená implementace jednoduše upravitelná pro další použití. Tato implementace slouží jako ukázka a zdroj k úpravě pro konkrétní použití knihovny pro OS Android. Pro serverovou část je také možné použít jinou platformu, ale je nutné dodržení dané specifikace metod, které musí API poskytovat pro správnou funkčnost knihovny. Dokumentace serverových metod s popisem, jak mají metody pracovat, je součástí příloh této práce.

Samotná implementace serverové části vychází z postupu založení nového projektu a jeho propojení s databází popsaného v teoretické části této práce. Když je projekt úspěšně propojený s databází, je možné začít implementovat jednotlivé modely a jejich řadiče (controller). V implementaci serverové části je vždy pro každou tabulku nutné vytvořit jeden řadič. Takže pokud vývojář vytvořil pomocí knihovny tři tabulky v mobilní aplikaci, je nutné mít na serverové části implementovány tři modely a k nim tři řadiče. Serverová implementace má však ještě o dva řadiče více. První řadič poskytuje metody pro registraci a přihlášení uživatele a druhý metody pro tabulku smazaných záznamů, která je zahrnuta nehledě na počet tabulek databáze. Řadič pro uživatele implementuje pouze dvě metody. POST metodu pro registraci uživatele (viz Ukázka kódu 11) a GET metodu pro jeho přihlášení.

```
public TokenDTO POST(RegistrationDTO model)
{
    using (var repository = new DiplomkaRepository(_databaseConnectionString))
    {
        if(model == null || model.Email == null || model.Pass == null)
        {
            throw new ServerException("No such method");
        }
        var users = repository.CarCashUsers.
                    FirstOrDefault(x => x.Email == model.Email);
        if(users == null)
        {
            var user = new CarCashUser()
            {
                Email = model.Email,
                Pass = model.Pass,
                Token = MakeToken(model.Email)
            };
            repository.CarCashUsers.Add(user);
            repository.SaveChanges();
            return new TokenDTO() { Token = user.Token };
        }
        else
        {
            throw new UserExistsException("Registration exists");
        }
    }
}
```

Ukázka kódu 11: Metoda POST poskytující registraci uživatele

Pro registraci uživatele se jeho údaje odesílají v těle dotazu. Při registraci jsou údaje součástí URL adresy. Obě tyto metody vracejí v těle odpovědi přístupový token. Ten si knihovna uloží a následně ho odesílá v hlavičce dotazů. Všechny ostatní metody, které poskytuje API, již potřebují ověření uživatele. V každé metodě se tedy nejprve provádí kontrola přihlášení. Pokud kontrola neprojde v pořádku, server vrátí chybovou odpověď. Toto ale víceméně knihovna nepřipouští, jelikož na její straně dochází vždy také k ověření, zda je uživatel přihlášen. Pro jistotu je to však ošetřeno i na straně serveru.

Řadič pro tabulku smazaných záznamů má pouze jednu metodu. Jedná se o metodu POST a slouží pro nahrávání smazaných záznamů na server. Navržená struktura dat (UUID záznamu, název tabulky, atd.) je odesílána v těle dotazu. Řadič tato data příjme a vykoná příslušné operace.

Řadiče pro vlastní definované tabulky jsou naprosto stejné, pouze pracují s jinými modely, které korespondují s danou tabulkou. Pokud se tedy přidává tabulka, je nutné vytvořit příslušné modely, nakopírovat původní řadič a upravit v něm pouze modelové třídy. Ve složce *Models* jsou modely, které korespondují se serverovou databází. Ve složce *API* je jeden soubor obsahující další modelové třídy, které jsou

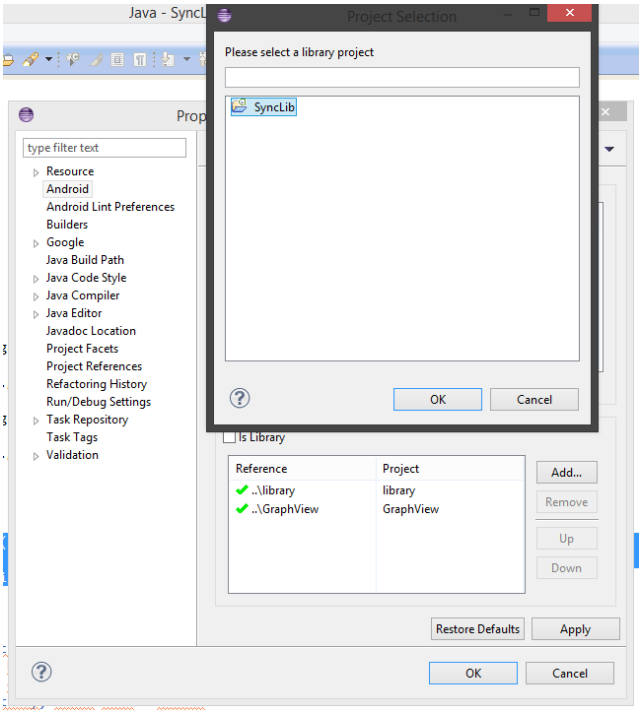
využívány pro příjem a odesílání dat z tabulek. Toto je z důvodu, že modelová třída pro korespondenci s databází obsahuje pouze proměnné neboli atributy tabulky. Modelové třídy pro komunikaci serverové části a knihovny obsahují deklarace názvů proměnných, které jsou využity při JSON reprezentaci. Dále jsou data odesílána jako pole objektů. Pro každý řadič jsou zde tedy tři modely. První model definuje podobu jednoho objektu neboli tabulky. Další model vytváří seznam objektů prvního modelu. Tento model je využíván pro nahrávání dat na server. Poslední model rozšiřuje předchozí model o seznam objektů s modelem smazaného záznamu. Tento model je využíván k odesílání nových a smazaných dat do knihovny.

Samotný řadič má opět pouze dvě metody. Metoda POST slouží pro nahrávání dat na server. Tato metoda přijímá data ve svém těle. Metoda sama vyhodnotí, zda se jedná o nový záznam, nebo o upravený záznam, a provede příslušnou operaci s databází. Rozpoznávání typu operace je umožněno díky využití UUID pro identifikaci záznamů. Metoda GET slouží k odesílání dat ze serveru. Tato metoda přijímá v URL adrese čas poslední synchronizace. Na základě přijatého času tato metoda vrátí novější záznamy a záznamy, které byly od poslední synchronizace smazány.

2.4 Použití knihovny pro OS Android

Knihovna je k této práci přiložena na CD ve dvou verzích, sestavená v .jar souboru a jako celý projekt. Doporučuje se použití .jar souboru. S tímto návodem je použití knihovny jednoduché. Celý projekt slouží pro případné vlastní úpravy nebo ke zkoumání, jak knihovna funguje.

Prvním krokem je založení nového projektu, ve kterém bude knihovna využívána. Toto zde vysvětlováno nebude, protože postup založení nového projektu byl popsán v bakalářské práci. Když je tedy vytvořený nový projekt, je možné přistoupit k použití knihovny. Pokud je použit .jar soubor, je nutné ho nakopírovat do složky *libs* v novém projektu. Ve vývojovém prostřední je dále nutné obnovit projekt a knihovna by měla již být automaticky přiložena. A nyní je možné její použití. Pokud je použit vlastní projekt knihovny, je nutné ho importovat do aktuálního pracovního adresáře. To se provede následovně: File –> Import –> Android –> Existing Android Code Into Workspace. V následující tabulce se vybere umístění projektu a zvolí se Finish. Projekt knihovny bude přidán do pracovního adresáře a je nutné ho propojit s novým projektem. Klikem pravým tlačítkem myši na nový projekt je nutné zvolit Properties. Zde je nutné vybrat kartu Android a dole zvolit tlačítko Add. Ve výběru knihoven bude pouze jedna (viz Obrázek 8). Tento projekt je nutné označit a přidat. Nyní je knihovna přiložena a může být použita úplně stejně jako v případě .jar souboru.



Obrázek 8: Přidání projektu jako knihovny

Prvním krokem k úspěšné implementaci knihovny je její inicializace při startu a uvolnění při vypínání aplikace. V nově vytvořené aplikaci je nutné vytvořit třídu, která bude dědit od *Application*. Jedná se o komponentu OS Android, která reprezentuje celou aplikaci. V této třídě je nutné přepsat metody *onCreate* a *onTerminate*. V těchto metodách se provede inicializace a uvolnění zdrojů knihovny. (viz Ukázka kódu 12)

```
public class CarCashApplication extends Application
{
    @Override
    public void onCreate()
    {
        SyncLib.getInstance().initializeSyncLib(this,
          "https://leos-diplomka-production.azurewebsites.net/api/",
           true);
        super.onCreate();
    }

    @Override
    public void onTerminate()
    {
        SyncLib.getInstance().disposeSyncLib();
        super.onTerminate();
    }
}
```

Ukázka kódu 12: Inicializace knihovny

Tuto třídu je nutné definovat v manifestu aplikace. Konkrétně přidáním parametru android:name do objektu application. Hodnota tohoto parametru musí obsahovat název balíčku, kde se vytvořená třída nachází, a název vytvořené Application třídy. Tímto je zaručena správná inicializace knihovny.

Dále je nutné přidat do manifestu aplikace povolení pro využívání služeb mobilního zařízení. Konkrétně je nutné povolit přístup na internet a zjišťování stavu připojení k internetu. Dále je nutné přidat do application objektu metadata, která ponesou název a verzi lokálního souboru s databází. Důležité také je přidat definice služeb do manifestu aplikace. Bez jejich přidání by synchronizace nefungovala. Ukázkový manifest aplikace se všemi potřebnými definicemi se nachází níže (viz Ukázka kódu 13).

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="aXer.BK.SledovaniNakladu"
    android:versionCode="3"
    android:versionName="1.2" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <uses-permission
         android:name="android.permission.INTERNET" />
    <uses-permission
         android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/logo"
        android:label="CarCash"
        android:theme="@style/Theme.MyTheme"
        android:name="aXer.BK.SledovaniNakladu.CarCashApplication"       >

        <meta-data android:name="AA_DB_NAME" android:value="CarCash.db" />
        <meta-data android:name="AA_DB_VERSION" android:value="1" />

        <service android:name="cz.dostalleos.synclib.service.
                               UploadItemIntentService"/>
        <service android:name="cz.dostalleos.synclib.service.
                              GetItemIntentService"/>

        <activity
            android:label="CarCash"
            android:name=".SledovaniNakladuAutomobiluActivity"
            android:theme="@android:style/Theme.NoTitleBar"
            android:configChanges="keyboardHidden|orientation" >
        </activity>

    </application>
</manifest>
```

Ukázka kódu 13: Ukázkový manifest aplikace s potřebnými definicemi

Dalším nezbytným krokem je definování modelových tříd, které budou reprezentovat tabulky databáze. Každá taková třída musí dědit z třídy Model, která se nachází v knihovně. Tím jsou poskytnuty metody pro ukládání, mazání a načítání záznamů z databáze. Nad definici třídy se přidá anotace, která definuje název dané tabulky. Nad jednotlivé proměnné se přidá také anotace, která definuje název atributu. Tyto anotace jsou povinné. Knihovna by měla podporovat všechny základní datové typy. K jednotlivým proměnným lze přidávat další parametry v anotaci, které umožňují nastavení parametrů jednotlivým atributům. Většina možností je vidět v ukázce (viz Ukázka kódu 4).

Pro vytvoření záznamu do dané tabulky stačí pouze vytvořit instanci dané třídy a naplnit ji daty pomocí metod pro nastavení proměnné nebo pomocí konstruktoru. Pokud je však definován konstruktor s parametry, je nutné definovat i konstruktor bez parametrů. Ve všech konstruktorech je nutné volat metodu *super*. Defaultní konstruktor je totiž využíván knihovnou. Když je instance s daty vytvořená, je možné zavolat metodu *save*. Ta provede uložení záznamu do databáze a pokus o odeslání záznamu na server.

```
CarCashTank tankDb = new CarCashTank();
tankDb.setAccountId(aRowId);
.
.
.
tankDb.setTypeTank(tankTypee);
long id = tankDb.save();
```

Ukázka kódu 14: Vytvoření záznamu v databázi

Pro získání záznamu se používá následující příkaz.

```
List<CarCashTank> tanks = new ArrayList<CarCashTank>();
tanks = new Select().from(CarCashTank.class).
       where("AccountId = ?", aRowId).orderBy("Id DESC").execute();
```

Ukázka kódu 15: Získání záznamů z databáze

Jedná se o obdobu klasického SQL příkazu. Metodě *from* se předává třída reprezentující tabulku, ze které je požadováno získání záznamu. Metodě *where* se předává podmínka pro selekci záznamů. Nejprve se uvádí v textovém řetězci název atributu a následně jeho požadovaná hodnota. Dále je možné použít metodu *orderBy*, které se předává textový řetězec s názvem atributu a typem řazení (ASC, DESC), podle kterého mají být záznamy řazeny. Pokud je požadován pouze jeden záznam, volá se nakonec metoda *executeSingle*. Pokud je požadováno více záznamů, volá se metoda *execute*, která vrátí seznam se záznamy. Mazání záznamu je možné pouze po jeho získání pomocí předchozího návodu. Mazání se provádí voláním metody *delete*. Tato metoda záznam smaže z tabulky, přesune jej do automaticky vytvořené tabulky se smazanými záznamy a pokusí se ho odeslat na server.

Takto použitá knihovna dokáže pracovat i v offline modu a bude se pouze starat o lokální databázi. Pokud je vyžadována synchronizace, je nutné registrovat nebo přihlásit uživatele. Knihovna pro toto poskytuje metody a posluchače. V následující

ukázce je zobrazeno použití metod pro registraci (viz Ukázka kódu 16) a přihlášení uživatele (viz Ukázka kódu 17).

```
SyncLib.getInstance().userRegistration
    (email.getText().toString(), pass.getText().toString(),
    new OnRegistrationFinished()
    {
        @Override
        public void OnRegistrationRespond
                    (int code, String message, String token)
        {
             if(code == 1)
             {
                 //Registrace proběhla úspěšně
             }
             else
             {
                 //Registrace se nezdařila, v proměnné message je zpráva
             }
        }
     });
```

Ukázka kódu 16: Příklad použití metody k registraci uživatele

```
SyncLib.getInstance().userLogin
    (email.getText().toString(), pass.getText().toString(),
    new OnLoginFinished()
    {
        @Override
        public void OnLoginRespond
                    (int code, String message, String token)
        {
            if(code == 1)
            {
                //Přihlášení proběhlo úspěšně
            }
            else
            {
                //Přihlášení se nezdařilo, v proměnné message je zpráva
            }
        }
     });
```

Ukázka kódu 17: Příklad použití metody k přihlášení uživatele

Metoda *isUserLogged* poskytuje informace, zda je uživatel přihlášen. Metoda *userLogOut* provádí odhlášení uživatele.

Poslední metodou je *SyncDb*, která provádí kompletní synchronizaci (odeslání neodeslaných záznamů, kontrolu nových a smazaných záznamů na serveru). Této metodě je nutné předat posluchač, který hlídá dokončení synchronizace. Toto je možné

například využít k zobrazení stavu průběhu (třída ProgressBar) a k jeho skrytí po ukončení synchronizace.

Knihovna pro informování aplikace poskytuje i broadcast vysílání, pro které je možné registrovat v aplikaci posluchač. Nejprve je nutné posluchač inicializovat. Pro toto slouží následující metoda (viz Ukázka kódu 18).

```
private IntentFilter mSyncFinishedIntentFilter;
private BroadcastReceiver mSyncFinishedBroadcastReceiver;

private void initReceiver()
{
    // create intent filter
    mSyncFinishedIntentFilter = new IntentFilter();

    mSyncFinishedIntentFilter.addAction
                (SyncFinishedBroadcast.ACTION_SYNC_FINISHED);

    // create broadcast receiver
    mSyncFinishedBroadcastReceiver = new BroadcastReceiver()
    {
        @Override
        public void onReceive(Context context, Intent intent)
        {
              if(intent.getAction().equals
                        (SyncFinishedBroadcast.ACTION_SYNC_FINISHED))
              {
                   // synchronizace dokončena
              }
        }
    };
}
```

Ukázka kódu 18: Metoda k inicializaci posluchače

Registrace posluchače probíhá v metodě *onResume* dané aktivity (viz Ukázka kódu 19).

```
@Override
protected void onResume()
{
   super.onResume();
   registerReceiver(mSyncFinishedBroadcastReceiver,
                        mSyncFinishedIntentFilter);
}
```

Ukázka kódu 19: Registrace posluchače

Posluchač je také nutné odhlásit (viz Ukázka kódu 20).

```
@Override
protected void onPause()
{
    super.onPause();
    unregisterReceiver(mSyncFinishedBroadcastReceiver);
}
```

Ukázka kódu 20: Odhlášení posluchače

2.5 Použití serverové části

Pro použití serverové části je nutné si nakopírovat projekt, který se nachází na přiloženém CD ve složce: Zdrojové kódy/Zdrojové kódy serverové části – ukázka. Prvním krokem k úspěšné implementaci je úprava konfiguračního řetězce pro propojení s databází. Konfigurační řetězec se nachází v souboru Web.config. Tento řetězec je nutné upravit dle konkrétního serveru s databází. Konfigurační řetězec je popsán v teoretické části této práce v kapitole 1.5.3.

Dále je nutné vytvořit v databázi příslušné tabulky. Tabulka v serverové databázi musí mít stejný název a musí obsahovat všechny atributy definované v knihovně, a to se stejným názvem a datovým typem. Dále je nutné tabulku rozšířit o následující atributy: *Id* (autoinkrement, int), *UUID* (String), *UserId* (int), *UpdateTime* (DateTime), *UploadedTime* (DateTime). Tyto atributy slouží pro správnou funkčnost synchronizace. Bez jejich definování by aplikace nefungovala. Stejně je nutné definovat model této tabulky v samotné implementaci serverové části ve složce *Models*. Název třídy musí být stejný jako název tabulky. Třída musí obsahovat proměnné se stejnými názvy a datovými typy jako atributy dané tabulky (viz Ukázka kódu 21). Každý tento vytvořený model je nutné přidat do dvou tříd ve složce *Repositories*. První třídou je DiplomkaContext.cs a druhou je DiplomkaRepository.cs. Přidání je jednoduché, stačí opět pouze zkopírovat uvedený příkaz v ukázkové aplikaci a přepsat modelovou třídu.

```
public class CarCashAccount
{
    public int Id { get; set; }
    public String UUID { get; set; }
    public int UserId { get; set; }
    public String Mark { get; set; }
    .
    .
    .
    public String Note { get; set; }
    public DateTime UpdateTime { get; set; }
    public DateTime UploadedTime { get; set; }
}
```

Ukázka kódu 21: Modelová třída pro databázi

Dále ve třídě UsersModels.cs ve vnořené složce *API* je nutné vytvořit ke každé tabulce tři modely (viz Ukázka kódu 22).

```
[DataContract]
public class CarCashAccountDTO
{
      [DataMember(Name = "UUID")]
      public string Uuid { get; set; }
      [DataMember(Name = "Mark")]
      public string Mark { get; set; }
      .
      .
      .
      [DataMember(Name = "Note")]
      public string Note { get; set; }
      [DataMember(Name = "updateDate")]
      public DateTime UpdateDate { get; set; }
}

[DataContract]
public class CarCashAccountsUploadDTO
{
      [DataMember(Name = "CarCashAccount")]
      public List<CarCashAccountDTO> Items { get; set; }
}

[DataContract]
public class CarCashAccountsDTO
{
      [DataMember(Name = "CarCashAccount")]
      public List<CarCashAccountDTO> Items { get; set; }
      [DataMember(Name = "CarCashAccountDeleted")]
      public List<ItemDeletedDTO> ItemsDeleted { get; set; }
}
```

Ukázka kódu 22: Modelové třídy pro komunikaci

První model reprezentuje opět danou tabulku, ale nyní musí obsahovat všechny atributy definované v knihovně a následující atributy: *UUID* (String) a *updateTime* (DateTime). Druhý model obsahuje pouze seznam objektů předchozího modelu. Deklarovaný název musí odpovídat názvu tabulky. Třetí model je stejný jako předchozí a přidává navíc seznam s objekty smazaných záznamů. Název tohoto seznamu se skládá z názvu dané tabulky a slova Deleted.

Předposledním krokem je nakopírování řadičů pro každou tabulku. Název řadiče musí být stejný jako název tabulky. V metodách daného řadiče se pouze změní příslušné modely.

Posledním krokem je úprava řadiče DeletedItemController.cs. Zde se nakopírují další bloky case v příkazu switch. Počet bloků case koresponduje s počtem definovaných tabulek. Každý blok case má podmínku s názvem tabulky a v jednotlivých blocích se upraví opět pouze modelové třídy.

Takto upravená implementace serveru může být zkompilována a nahrána na server. V této práci bylo využíváno systému správy verzí společnosti GitHub.

Nahrávání implementace probíhalo jednodušeji. Celý projekt se nahrál do repositáře na server GitHub. Repositář byl propojen s Windows Azure, během několika minut byla implementace automaticky nahrána na server a byla již funkční.

2.6 Aplikace využívající knihovnu a serverovou část

Pro provedení testování synchronizace byla vytvořená knihovna implementována do aplikace pro správu provozních výdajů automobilu, která byla vytvořena v rámci bakalářské práce a diplomového projektu. Pro účely testování byla také vytvořena příslušná implementace serverové části.

Implementace knihovny probíhala dle kroků popsaných v kapitole 1.5. Během implementace do této aplikace bylo prováděno opravování chyb a zlepšování použití knihovny tak, aby bylo co nejjednodušší. Výsledkem je lehce použitelná knihovna, která v testovací aplikaci funguje bezproblémově a spolehlivě. Původní aplikaci bylo nutné rozšířit o možnost registrace a přihlášení uživatele. K tomu byla navržena nová aktivita, která je zobrazena při prvním spuštění aplikace. V této aktivitě je možné vytvořit profil vozidla a používat aplikaci v offline modu bez synchronizace. Pokud však uživatel chce využívat synchronizaci, musí se nejprve registrovat. Pokud má již účet vytvořený, musí se přihlásit. Pokud se uživatel přihlásí k již existujícímu účtu, jeho data jsou stažena do mobilního zařízení ze serveru. Aplikace nyní provádí kompletní synchronizaci při každém startu. Synchronizaci je možné vyvolat z menu aplikace (viz Obrázek 9). Při vytváření nového záznamu je provedeno odeslání ihned po jeho uložení. V aplikaci je také implementováno rozlišení odeslaných záznamů od neodeslaných, a to barvou pozadí příslušného záznamu. Světle modré pozadí značí, že je záznam odeslán. Tmavě modré pozadí značí, že ještě není odeslán (viz Obrázek 10).

Obrázek 9: Ukázková aplikace - menu

Aby byla aplikace plně kompatibilní a umožňovala update ze starší verze, bylo nutné původní databázi ze starší verze překopírovat do nové databáze. Toto bylo nutné, protože původní verze aplikace pracovala s databází pomocí klasického přístupu v OS Android. Knihovna však využívá ORM, takže bylo nutné databázi překopírovat a původní odstranit. S tímto nebyl žádný problém, překopírování funguje v pořádku a rychle.

Obrázek 10: Ukázková aplikace - záznamy

Pokud se uživatel rozhodne aplikaci nejprve používat v offline modu bez přihlášení nebo pokud provedl aktualizaci staré aplikace, tak po přihlášení nebo registraci budou všechny záznamy automaticky synchronizovány.

Serverovou aplikaci bylo nutné rozšířit na poskytování metod pro všechny tabulky mobilní aplikace. V aplikaci se nacházejí 4 tabulky, jedna tabulka s vozidly a další tři tabulky s různými typy záznamů. Nejprve tedy bylo nutné vytvořit na serveru databázi, která koresponduje s lokální databází. Následně bylo nutné vytvořit příslušné modelové třídy na straně serveru a příslušné řadiče. Takto upravená implementace serverové části byla nahrána na server a otestována. Zdrojové kódy samotné aplikace pro OS Android a celé implementace serverové části jsou přiloženy na CD. Serverová část běží na adrese http://leos-diplomka-production.azurewebsites.net/. Zde je dostupná webová aplikace pro zobrazení uživatelských dat. Na adrese http://leos-diplomka-production.azurewebsites.net/api/ pak běží webové API, které poskytuje všechny

metody pro tuto aplikaci. Dokumentace této implementace je dostupná ve webové aplikaci pod záložkou Nápověda k API.

2.6.1 Implementace webové aplikace

Implementace webové aplikace na platformě .NET je obdobná jako implementace webového API. Je také založena na MVC návrhovém vzoru a vše je možné implementovat v jednom projektu na jednom serveru. Každá HTML stránka ve webové aplikaci musí mít implementovány dvě komponenty, řadič a pohled. Volitelnou částí je model. Každý model může využívat více stránek. Řadič slouží pro vnitřní logiku stránky. V řadiči se například získávají data z databáze a jsou v daném modelu předána pohledu, ve kterém je vygenerována výsledná HTML stránka.

V řadiči se také zpracovávají odeslané formuláře. V modelu se vytvoří pomocí HTML tagů a jazyka ASP formulář. V pohledu se definuje, který řadič a která metoda má formulář zpracovat. Dále se jednotlivým textovým polím nastaví, do které proměnné daného modelu mají být přiřazené. Následně je po odeslání formuláře daný model předán metodě v definovaném řadiči. Ten se postará o jeho zpracování a na základě předaných dat provede nějakou operaci. Takto je v této webové aplikaci vyřešeno přihlášení uživatelů. Byl vytvořen model, který obsahuje dvě proměnné, e-mail a heslo. Uživatel tedy vyplní tyto údaje a potvrdí formulář. Pohled toto přesměruje na daný řadič a ten zkontroluje zadané údaje s databází uživatelů. Pokud údaje souhlasí, uživatel je přihlášen a řadič přesměruje uživatele na hlavní stránku. Pokud zadané údaje nesouhlasí, formulář je obnoven. Webová aplikace si po nějakou dobu přihlášení pamatuje. Přihlášení je zaručeno pomocí cookies, do kterých aplikace uloží uživatelův e-mail. Této cookie je následně využíváno na všech stránkách aplikace ke zjišťování, která data se mají načíst z databáze. Z aplikace je možné se odhlásit. Odhlášení provede smazaní uložené cookie, uživatel je tím odhlášen a je nutné se znovu přihlásit.

Ošetření přístupnosti stránek je naprosto jednoduché. Stačí přidat Authorize nad definici řadiče (viz Ukázka kódu 23).

```
[Authorize]
public class WebAccountController : Controller
{
.
.
.
}
```

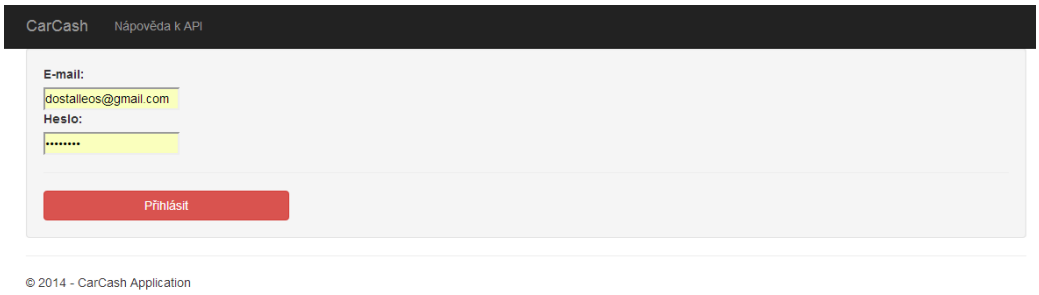
Ukázka kódu 23: Příklad použití Authorize

Tím je omezena jeho přístupnost pouze na autorizované uživatele, tedy přihlášené. Pokud nějaký uživatel přistupuje ke stránce, která má takto nastavený řadič, je vždy přesměrován na přihlašovací formulář (viz Obrázek 11). Ten musí být definován v souboru Web.config konfiguračním řetězcem (viz Ukázka kódu 24).

```
<authentication mode="Forms">
      <forms loginUrl="~/WebAccount/Login" timeout="2880" />
</authentication>
```

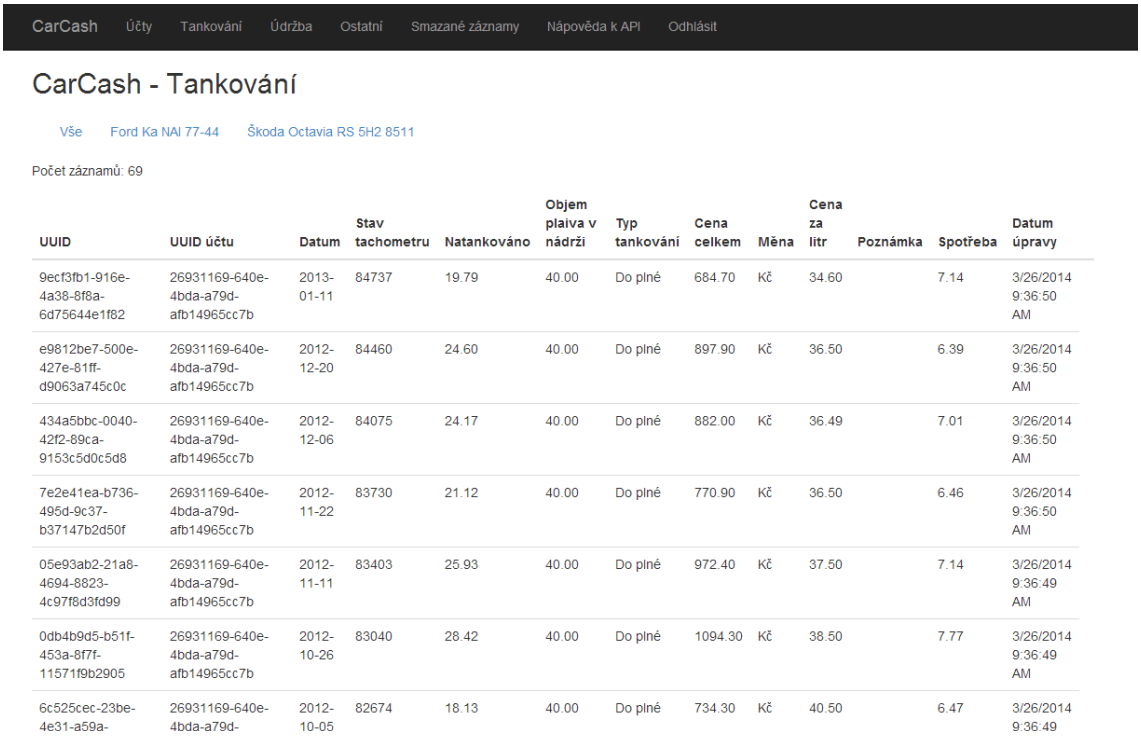
Ukázka kódu 24: Konfigurace přihlašovacího formuláře

Ve výchozím stavu má každý řadič metodu *Index*. Jedná se o výchozí zobrazení stránky. V řadiči dále mohou být metody pro zobrazení detailu, obrazovky pro úpravy atd. Ke každé takové metodě je nutné vytvořit daný pohled. Název metody a pohledu musí být stejný. Ke každému řadiči je vytvořena složka s jeho pohledy. Název této složky je stejný jako název řadiče.



Obrázek 11: Přihlašovací formulář

V této aplikaci je využito pouze základních pohledů pro zobrazení dat. Aby webová aplikace korespondovala s mobilní, je zde pět řadičů pro zobrazování dat. Každý řadič má pouze metodu *Index* a k ní příslušný pohled. V řadiči se pouze načítají příslušná data z databáze dle přihlášeného uživatele. Tato data jsou následně předána pohledu a tam je vygenerována HTML stránka (viz Obrázek 12). Na stránce je tabulka se všemi daty z dané databázové tabulky. Záznamy v kategoriích Tankování, Údržba a Ostatní je možné filtrovat dle vozidel. To znamená, že v řadičích zmíněných kategorií dochází k načítání nejen dat z příslušné tabulky, ale načítají se také údaje o vozidlech z jiné tabulky.



Obrázek 12: Výsledná webová aplikace

3 Závěr

Výsledkem této práce je synchronizační balík, který umožňuje automatickou synchronizaci mobilní databáze se serverovou databází. První vytvořenou částí synchronizačního balíku je klientská aplikace v podobě knihovny. Tato knihovna umožňuje vývojáři jednoduše definovat lokální databázi a poskytuje metody pro registraci a přihlášení uživatelů. Knihovna zajistí automatickou synchronizaci záznamů se serverovou databází bez zásahu vývojáře a uživatele. Knihovna ošetřuje konfliktní stavy během synchronizace.

Druhou částí balíku je implementace serverové části v jazyce ASP.NET. Tato ukázková implementace serverové části slouží pro odvození konkrétní implementace serveru pro danou aplikaci. Přizpůsobení této ukázkové implementace je jednoduché. Vývojář musí pouze vytvořit příslušné modely a řadiče dle postupu popsaného v této práci. V ukázkové implementaci jsou podrobné komentáře pro snadné odvození nové implementace.

Celý tento balík byl v této práci použit na již existující mobilní aplikaci pro správu provozních výdajů automobilu. Test synchronizace byl proveden mezi dvěma zařízeními s OS Android a nainstalovanou aplikací pro správu provozních výdajů. Ke kontrole synchronizace byla ještě využívána vytvořená webová aplikace, která zobrazuje uživatelova data.

Byly otestovány všechny problematické situace jako nedostupnost internetového připojení, smazání již smazaného záznamu na nesynchronizovaném zařízení a úprava jednoho záznamu na dvou zařízeních. Navržená synchronizace fungovala bez problémů a bez zbytečně přenášených dat. Přenos dat probíhá zabezpečenou verzí protokolu HTTP. Je tedy myšleno i na bezpečnost. Uživatelské heslo je přenášeno a uchováváno v podobě MD5 haše. Uživatel se při synchronizaci identifikuje přiděleným tokenem, který je přenášen v hlavičkách dotazů.

Tato práce splnila všechny body zadání. Synchronizační balík je do budoucna možné rozšířit o další ukázkové implementace serverové části na jiných platformách. Dalším užitečným rozšířením knihovny pro OS Android by bylo navržení a implementování metod pro automatické přenesení původní databáze vytvářené klasickým přístupem v OS Android.

Seznam použité literatury

[1] DOSTÁL, Leoš. 2012. *Aplikace pro sledování nákladů provozu automobilu pro OS Android*. Liberec, Bakalářská práce. Technická univerzita v Liberci. Vedoucí práce Ing. Přemysl Svoboda.

[2] DOSTÁL, Leoš. 2012. Rozšíření a*plikace pro sledování nákladů provozu automobilu pro OS Android*. Liberec, Dostupné z: Přiložené CD. Diplomový projekt. Technická univerzita v Liberci. Vedoucí práce Ing. Přemysl Svoboda.

[3] MLNÁŘÍK, David. *Synchronizace databází pro platformu Android* [online]. 2013 [cit. 2014-05-07]. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce doc. RNDr. Vlastislav Dohnal, Ph.D.. Dostupné z: http://is.muni.cz/th/324590/fi_m/.

[4] PARDO, Michael. *ActiveAndroid* [online]. 2014 [cit. 2014-05-07]. Dostupné z: http://www.activeandroid.com/.

[5] Services. *Android Developers* [online]. 2014 [cit. 2014-05-07]. Dostupné z: http://developer.android.com/guide/components/services.html.

[6] MURPHY, Mark L. 2011. *Android 2: průvodce programováním mobilních aplikací*. Vyd. 1. Brno: Computer Press, 375 s. ISBN 978-80-251-3194-7.

[7] ASP.NET Web API. *Microsoft Developer Network* [online]. 2014 [cit. 2014-05-07]. Dostupné z: http://msdn.microsoft.com/en-us/library/hh833994(v=vs.108).aspx.

[8] *NuGet Gallery* [online]. 2014 [cit. 2014-05-07]. Dostupné z: http://www.nuget.org/.

[9] MIKOLUK, Kasia. JSON vs XML: How JSON Is Superior To XML. *Udemy Blog* [online]. 2013 [cit. 2014-05-07]. Dostupné z: https://www.udemy.com/blog/json-vs-xml/.

[10] PARDO, Michael. ActiveAndroid. In: *GitHub* [online]. 2014 [cit. 2014-05-07]. Dostupné z: https://github.com/pardom/ActiveAndroid.

[11] UUID. *Android Developers* [online]. 2014 [cit. 2014-05-07]. Dostupné z: http://developer.android.com/reference/java/util/UUID.html.

[12] IntentService. *Android Developers* [online]. 2014 [cit. 2014-05-07]. Dostupné z: http://developer.android.com/reference/android/app/IntentService.html.

Seznam příloh

Příloha A

Dokumentace ukázkové implementace serveru

Dokumentace ukázkové implementace serveru

1. Ošetření chybových stavů

Zde je ukázková specifikace chybové odpovědi serveru. Jednotlivé chybové kódy a zprávy jsou popsány u jednotlivých dotazů. Globální chyba serveru má id = 10.

```
HTTP/1.1 200 OK

{
  "error":
  {
    "id": 110,
    "message": "The request is invalid."
  }
}
```

Parametr	Volitelný	Více
id	ne	Identifikátor chyby.
message	ano	Zpráva o chybě

2. Registrace uživatele

Specifikace dotazu pro registraci uživatele:

```
POST /User HTTP/1.1

Content-Type: application/json

{

  "email": "dostalleos@gmail.com",

  "pass": "123456"

}
```

Parametr	Volitelný	Více
email	ne	E-mail uživatele
pass	ne	MD5 haš hesla

Specifikace odpovědi pro registraci uživatele:

```
HTTP/1.1 200 OK

{

  "access_token": "sdfa66fasdf2asdf"

}
```

Parametr	Volitelný	Více
access_oken	no	Token uživatele

Chyba:

id	Popis
110	Registrace již existuje

3. Přihlášení uživatele

Specifikace dotazu pro přihlášení uživatele:

```
GET /User?email=dostalleos@gmail.com&pass=123456 HTTP/1.1
```

Parametr	Volitelný	Více
email	ne	E-mail uživatele
pass	ne	MD5 haš hesla

Specifikace odpovědi pro přihlášení uživatele:

```
HTTP/1.1 200 OK

{

  "access_token": "fj3AZjhIhB39AkRT"

}
```

Parametr	Volitelný	Více
access_token	ne	Token uživatele

Chyba:

id	Popis
100	Uživatel neexistuje / heslo není správně

4. Získání záznamů z tabulky Item

Zde je dotaz na získání dat z tabulky Item, která má následující vývojářem definované atributy: firstName, lastName, email. Ostatní atributy musí poskytovat API. Tento dotaz je nutné vytvořit pro všechny tabulky definované vývojářem a dále je nutné upravit dané atributy. Tato metoda vrátí všechny záznamy z dané tabulky, které byly uloženy až po přijatém čase. Metoda také vrací všechny smazané záznamy dané tabulky, které byly smazány až po přijatém čase.

```
GET /Item?date=1.1.2013 HTTP/1.1

Authorization: asdfasdf
```

Parametr	Volitelný	Více
Authorization	ne	Token uživatele
date	ne	Datum poslední aktualizace tabulky

Odpověď:

```
HTTP/1.1 200 OK

{
  "Item": [
   {
     "UUID": "afsdf3af25af",
     "firstName": "Leos",
     "lastName": "Dostal",
     "email": "dostalleos@gmail.com",
     "updateDate": "2014-05-07T11:57:05.1658522+00:00"
   },
   {
     "UUID": "afsdf3af25af",
     "firstName": "Jan",
     "lastName": "Dostal",
     "email": "dostaljan@gmail.com",
     "updateDate": "2014-05-07T11:57:05.1658522+00:00"
   },
   ...
   ],
  "ItemDeleted": [
   {
     "UUID": "asfadf5asf4"
   },
   {
     "UUID": "adfasdfasdf5"
   },
   ...
```

```
    ]
}
```

Parametr	Volitelný	Více
UUID	ne	UUID záznamu
firstName	ano	
lastName	ano	
email	ano	
updateDate	ne	Čas poslední editace záznamu

Chyba:

id	Popis
100	Uživatel neexistuje

5. Přijímání záznamů do tabulky Item

Zde je uveden dotaz pro přijímání záznamů do tabulky Item, která má následující vývojářem definované atributy: firstName, lastName, email. Ostatní atributy musí poskytovat API. Tento dotaz je nutné vytvořit pro všechny tabulky definované vývojářem a také je nutné upravit definované atributy. Metoda provede přijetí záznamů a rozhodne pro každý záznam, zda se jedná o operaci vložení, nebo aktualizace. Metoda přidělí záznamu čas přijetí na server a uloží ho do serverové databáze.

```
POST /Item HTTP/1.1

Content-Type: application/json

Authorization: asdfasdf

{

  "Item": [

   {

      "UUID": "afsdf3af25af",

      "firstName": "Leos",

      "lastName": "Dostal",

      "email": "dostalleos@gmail.com",

      "udpateDate": "2014-05-07T12:14:39.3394934+00:00"

    },

    {

      "UUID": "afsdf3af25af",

      "firstName": "Jan",

      "lastName": "Dostal",

      "email": "dostalJan@gmail.com",

      "udpateDate": "2014-05-07T12:14:39.3394934+00:00"

    },

    ...

   ]
```

```
}
```

Parametr	Volitelný	Více
Authorization	ne	Token uživatele
UUID	ne	UUID záznamu
firstName	ano	
lastName	ano	
email	ano	
updateDate	ne	Čas poslední editace záznamu

Odpověď:

```
HTTP/1.1 200 OK
```

Chyba:

id	Popis
100	Uživatel neexistuje

6. Přijímání smazaných záznamů

Zde je uveden dotaz pro přijímání smazaných záznamů z automaticky vytvořené tabulky DeletedItems. Tato specifikace musí být dodržena přesně. Metoda příjme záznamy, každému záznamu přidělí čas nahrání na server a uloží ho do serverové databáze.

```
POST /DeletedItem HTTP/1.1
Content-Type: application/json
Authorization: asdfasdf
{
  "DeletedItem": [
   {
      "UUID": "asfdasdfaf5afas",
      "ItemTable": "Item",
      "ItemUUID": "asdfas5dfas5fafasdf",
      "updateDate": "2014-05-07T12:25:45.0127812+00:00"
    },
    {
      "UUID": "asfd55dfaf5afass",
      "ItemTable": "Item",
      "ItemUUID": "asdfgs545dfas5fafasdf",
      "updateDate": "2014-05-07T12:25:45.0127812+00:00"
    },
    ...
   ]
}
```

Parametr	Volitelný	Více
Authorization	ne	Token uživatele
UUID	ne	UUID záznamu
ItemTable	ne	Název tabulky smazaného záznamu
ItemUUID	ne	UUID smazaného záznamu
updateDate	ne	Čas smazání

Odpověď:

```
HTTP/1.1 200 OK
```

Chyba:

id	Popis
120	Tento záznam je již smazán
100	Uživatel neexistuje

Příloha B

CD-ROM

CD-ROM

Na přiloženém CD je následující struktura adresářů:

- Diplomová práce: Obsahuje text této práce ve formátu pdf.
- Sestavené soubory: Obsahuje instalační balíček aplikace CarCash a soubor sestavené knihovny SyncLib.
- Zdrojové kódy:
    - Zdrojové kódy – použití knihovny SyncLib: Ukázkový projekt aplikace pro OS Android, který zobrazuje využití knihovny SyncLib.
    - Zdrojové kódy aplikace CarCash: Zdrojové kódy aplikace CarCash, do které byla implementována vytvořená knihovna. Obsahuje i zdrojové kódy dalších použitých knihoven.
    - Zdrojové kódy knihovny SyncLib: Zdrojové kódy vytvořené knihovny SyncLib.
    - Zdrojové kódy serverové části – CarCash: Zdrojové kódy implementace serverové části pro mobilní aplikaci CarCash.
    - Zdrojové kódy serverové části – ukázka: Zdrojové kódy ukázkové implementace serverové části. Tento projekt slouží k rozšíření pro konkrétní implementaci.